Python编程语言

高级篇

零基础入门系列教程

作者: 魏明择

2025 年版

https://weimingze.com

目录

序

前言

达成目标：

- 成为 Python 编程语言专家，知晓Python全部的语法。
- 精通 Python 内部的工作原理
- 能够看懂Python官方文档中的全部内容
- 能够看懂标准库可第三方库的源码并进行分析
- 能够自己组织架构，实现框架的编写
- 能够编写出高效率的程序

适合人群

- 对编程感兴趣的人、编程初学者
- 全国计算机等级考试 - 二级《Python语言程序设计》考生
- 数据科学家和数据分析师
- 科研人员和工程师
- 自动化运维人员
- 游戏开发者
- 中小学教师

特点：

- 知识点深入解析
- 深入浅出，剖析python内部原理

内容（预告）

本课程将讲解如下内容：

- 异常
- 字节串和字节数组
- 文件（高级）
- 汉字编码
- 生成器和迭代器
- 迭代工具函数

- 函数式编程
- 函数（高级）
- python作用域
- 函数参数标注
- 闭包
- 装饰器
- 类和对象（高级）
- 面向对象（高级）
- 特性属性
- 内置函数重写
- 运算符重载

第二十章、异常

1. 异常

什么是错误

错误是指由于逻辑或语法等导致一个程序已无法正常执行的问题。

什么是异常

异常是程序执行过程中发生的错误时标识的一种状态，此状态下程序无法正常处理的原定的流程，此时程序不会在继续执行，而是通过特殊路径将错误返回到上层调用者（函数或方法），等待处理的过程称之为异常。

异常的作用:

用作信号, 通知上层调用者有错误产生需要处理。

异常示例

以下代码在输入成绩时输入: 3.14 就会引发 ValueError类型的错误!

```
score = int(input('请输入一个整数: '))
print('score:', score)

print('程序正常退出')
```

运行结果:

```
请输入一个整数: 3.14
Traceback (most recent call last):
  File "/Users/weimz/Desktop/chapter_backup/chapter_20/01_errors.py", line 8, in
<module>
    score = int(input('请输入一个整数: '))
            ~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: '3.14'

进程已结束，退出代码为 1
```

这种情况是程序中 int(x) 函数不能处理字符串 '3.14'，从而发出错误信号通知，并进入异常状态，并将错误信号传递给上层调用者处理。要出来此情况需要用到 try 语句。

try语句

作用：用于处理异常。

try语句分类：

1. try-except语句；
2. try-finally语句；
3. try-except * 语句(python 3.11新增,不讲)。

2. try-except语句

try-except语句

作用

偿试捕获异常，得到异常通知，将程序由异常流程转为正常流程并继续执行。

说明

1. 至少有一条except子句。
2. 如果没有匹配到任何错误类型，则程序的异常状态会继续下出,并向上层(调用处)传递。

语法

```
try:
    可能引发异常的语句
except 异常类型1 [as 变量1]:
    异常处理语句1
except 异常类型2 [as 变量2]:
    异常处理语句2
...
except:
    异常处理语句other
else:
    未发生异常语句
finally:
    最终语句
```

示例

```
# try-except语句示例

def sharing_apple(apple_count, person_count):
    ''' 将apple_count 个苹果分给 person_count 个人，并打印结果!'''
    try:
```

```
        person_count = int(person_count)  # 可能触发ValueError类型的错误
        result = apple_count / person_count  # 可能触发ZeroDivisionError和
TypeError类型的错误
        print('每个人分', result, '个苹果',)
    except TypeError:
        print('类型错了,苹果数必须是数字')
    except ZeroDivisionError:
        print('人数为零，不能做除法')

# 调用分苹果函数
try:
    # sharing_apple(10, 2)
    # sharing_apple('10', 2)
    # sharing_apple(10, 0)
    sharing_apple(10, '2.5')
    print('主模块中程序是正常状态')
except:
    print('最上层异常被捕获!')
print("程序正常执行并退出! ")
```

使用try 语句可以接受并处理 int(x) 函数 和 `/` 运算符引发的错误通知，并能将程序由异常状态转换为正常状态。

3. try-except语句的子句

try-except语句的子句

语法

```
try:
    可能引发异常的语句
except 异常类型1 [as 变量1]:
    异常处理语句1
except 异常类型2 [as 变量2]:
    异常处理语句2
...
except:
    异常处理语句other
else:
    未发生异常语句
finally:
    最终语句
```

说明:

- else子句在没有异常和跳转时，则执行其中的语句。

- finally子句在try语句执行完毕后一定执行其中的语句。
- 如果 finally 子句执行了 return, break 或 continue 语句，则被保存的异常会被丢弃，即转为正常状态。

示例

```
# try-except语句的子句示例

def sharing_apple(apple_count, person_count):
    ''' 将apple_count 个苹果分给 person_count 个人，并打印结果!'''
    try:
        person_count = int(person_count)  # 可能触发ValueError类型的错误
        result = apple_count / person_count  # 可能触发ZeroDivisionError和
TypeError类型的错误
        print('每个人分', result, '个苹果',)
    except TypeError:
        print('类型错了,苹果数必须是数字')
    except ZeroDivisionError:
        print('人数为零，不能做除法')
    else:
        print('正常流程!')
    finally:
        print('我一定会被打印!')

# 调用分苹果函数
# sharing_apple(10, 2)  # 正常流程
# sharing_apple('10', 2)  # TypeError
# sharing_apple(10, 0)
sharing_apple(10, '2.5')
print('主模块中程序是正常状态')

print("程序正常执行并退出! ")
```

上述示例中，else 子句会在try 内部没有引发异常是处理。

finally 子句在任何时候都一定会执行。

常见的异常类型

缩进代表继承关系。

```
BaseException
    Exception
        ArithmeticError
            ZeroDivisionError
        ImportError
            ModuleNotFoundError
        LookupError
            IndexError
```

```
            KeyError
        NameError
        ...
        OSError
            FileNotFoundError
        RuntimeError
            NotImplementedError
        StopIteration
        TypeError
        ValueError
    KeyboardInterrupt
```

Python 3.13 中全部的异常类型如下：

```
BaseException
    BaseExceptionGroup
        ExceptionGroup(BaseExceptionGroup, Exception)
    Exception
        ArithmeticError
            FloatingPointError
            OverflowError
            ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
            ModuleNotFoundError
        LookupError
            IndexError
            KeyError
        MemoryError
        NameError
            UnboundLocalError
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError
                ConnectionAbortedError
                ConnectionRefusedError
                ConnectionResetError
            FileExistsError
            FileNotFoundError
            InterruptedError
            IsADirectoryError
            NotADirectoryError
            PermissionError
            ProcessLookupError
            TimeoutError
        ReferenceError
        RuntimeError
            NotImplementedError
            PythonFinalizationError
            RecursionError
```



```
        StopAsyncIteration
        StopIteration
        SyntaxError
            IndentationError
                TabError
        SystemError
        TypeError
        ValueError
            UnicodeError
                UnicodeDecodeError
                UnicodeEncodeError
                UnicodeTranslateError
        Warning
            BytesWarning
            DeprecationWarning
            EncodingWarning
            FutureWarning
            ImportWarning
            PendingDeprecationWarning
            ResourceWarning
            RuntimeWarning
            SyntaxWarning
            UnicodeWarning
            UserWarning
    GeneratorExit
    KeyboardInterrupt
    SystemExit
```

练习

写一个函数get_score() 来获取学生输入的成绩（0～100的整数),如果输入出现异常，则此函数返回0，否则返回用户输入的成绩.

4. try-finally语句

作用

1. 通常用try-finally语句来做引发异常时必须要的处理的事情。
2. 无论异常是否发生，finally子句都会被执行。

语法

```
try:
    可能引发异常的语句
finally:
    最终语句
```

说明

- finally子句不可以省略。
- 一定不存在except子句语法。

示例

```
# try-finally语句示例

def write_nx_to_files():
    '''向文件写入n个x'''
    file = open('test.txt', 'w')
    try:
        number = int(input('请输入写入x的个数:'))
        file.write('x' * number)
    finally:
        file.close()
        print('文件已经关闭')


write_nx_to_files()
```

上述程序中，当 try 语句内引发异常时，finally 子句能够保证一定能关闭文件。但异常状态没有改变。错误信息依旧会向上传递给上层的 try-except 语句。如果没有 try-except 出来，则程序异常终止。

finally子句的特殊情况

- 终止异常的情况
- 如果 finally 子句执行了 return、break 或 continue 语句，则被保存的异常会被丢弃，即转为正常状态

5. try语句嵌套

try嵌套是指一个try语句嵌套在另外一个try语句内部使用。

示例

```
# try语句嵌套示例

def write_nx_to_files():
    '''向文件写入n个x'''
    try:
        file = open('test.txt', 'w')  # OSError
        try:
```

```
            number = int(input('请输入写入x的个数:'))  # ValueError
            file.write('x' * number)
        finally:
            file.close()
            print('文件已经关闭')
    except OSError:
        print('放弃存储!')
    except ValueError:
        print('输入数字有错，放弃存储!')

write_nx_to_files()
```

try嵌套时异常状态传递

先在内部 try 语句中的 except 子句中处理，如果内部 try 语句没有 except 子句或 except 子句没有对应的类型来捕获此异常，则此异常将传递到外层的try语句再进行处理，直接将异常状态传递到最外层的 try 语句为止。

6. raise语句

异常的作用:

用作信号通知，通知上层调用者有错误产生需要处理。

接收异常通知的语句

1. try-except语句。
2. try-except*语句。

发出异常通知的语句

1. raise语句。
2. assert语句。

作用

用于手动引发异常。发送错误通知给调用者。

做法

将一个错误对象放入系统中，让程序进入异常状态。

语法

```
raise 异常对象
# 或
raise 异常类型
# 或
raise  # 重新引发当前处理的异常
```

示例

```
# raise语句示例

def get_score():
    '''
    此函数返回用户输入的成绩，正常成绩在0~100范围，
    其他值触发ValueError类型的异常通知
    '''
    score = int(input("请输入您的成绩: "))
    if score < 0:
        err_obj = ValueError(f'成绩{score}是负数!')
        raise err_obj
    if score > 100:
        raise ValueError
    return score

try:
    score = get_score()
    print("学生的成绩是:", score)
except ValueError as err:
    print('成绩录入有错! err:', err)
    raise
```

上述程序中，当用户输入的数字小于零时，会引发 ValueError 类型的错误，并让程序进入异常处理流程。

当用户输入的数字大于 100 是，会用 没有参数的 ValueError() 构造函数创建一个错误对象，并让程序进入异常处理流程。

在 except ValueError as err: 子句中的 raise 语句是处理完 get_score() 函数内部的错误后，又将err绑定的错误再向上传递，并重新进入异常处理流程。

练习

写一个函数 get_age() 用来获取一个人的年龄信息，此函数规则用户只能输入1~140之间的整数，如果用户输入其它的数则直接引发ValueError类型的错误!

```
def get_age():
    ...  # 完成此处的代码
```

```
try:
    age = get_age()
    print("用户输入的年龄是:", age)
except ValueError as err:
    print("用户输入的不是1~140的整数，获取年龄失败!")
```

7. 自定义异常类型

python支持自定义异常类，这样可以丰富错误的类型，明确区分错误，提高代码的可读性，并可以分层处理错误信息。

异常类都必须继承自 BaseException 类。

BaseException类的示例代码

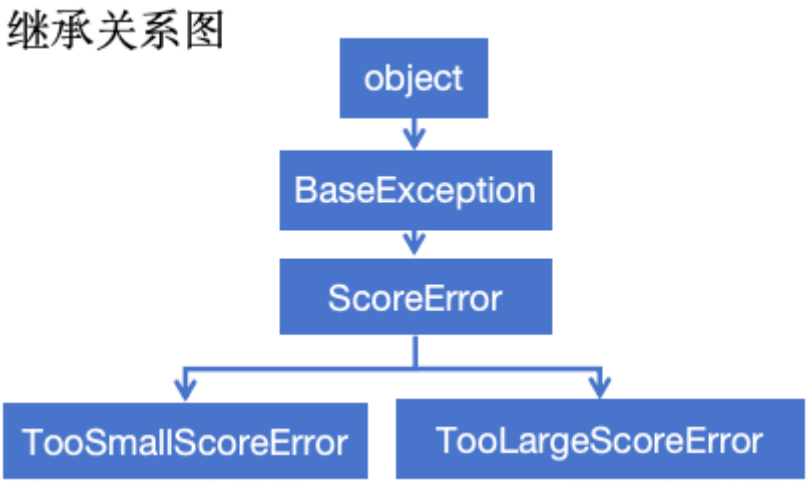
```
class BaseException(object):
    def __init__(self, *args):
        self.args = args
```

示例:

```
# 此示例示意自定义异常类型

class ScoreError(BaseException):  # 分数错误!
    def __init__(self, data):
        self.data = data
        super().__init__(data)

class TooSmallScoreError(ScoreError):  # 分数小于零的错误
    pass

class TooLargeScoreError(ScoreError):  # 分数太大的错误
    pass

def get_score():
    score = int(input("请输入您的成绩: "))
    if score < 0:
        err = TooSmallScoreError(f'{score}')
        raise err
    return score

try:
    score = get_score()
    print("学生的成绩是:", score)
except TooSmallScoreError as e:
    print('成绩太小了, 分数', e.data)
except ScoreError as e:
    print('成绩错了,e:', e.data)
```

上述三个类 ScoreError、TooSmallScoreError 和 TooLargeScoreError 类型都是自定义的异常类型。并且子类会属于父类的类型。

他们的继承关系如下图所示：



8. 内置常量__debug__

内置常量__debug__

__debug__是python的内置常量，用于判断运行模式。

__debug__的值

- True，处于调试模式。
- False，处于优化运行模式（项目上线时使用），如果 Python 以 -O 选项启动，则此常量为假值。

__debug__的作用

- 控制调试代码。
- 优化性能。

示例:

```
# __debug__ 特殊的内置常量示例

print('__debug__:', __debug__)
```

上述示例在运行时，根据运行的参数不同 __debug__ 变量的值也会不同。如下：

没有选项的运行，默认是调试运行：

```
% python3 08_debug_const_bool.py
__debug__: True
```

使用 `-O` 或 `-OO` 选项的运行，则认为是生成环境下的优化运行：

```
% python3 -O 08_debug_const_bool.py
__debug__: False
```

python运行时选项（常用）

- `-O`：
  - 启用基本优化模式，移除 assert 语句并将 **debug** 设置为 False。
- `-OO`：
  - 在 -O 的基础上进一步优化，除了移除 assert 语句，还会移除文档字符串。
- `-c`：
  - 直接执行命令行中指定的 Python 代码。
  - 例如：python -c "print('Hello, World!')"。
- `-m`：
  - 将模块作为脚本运行。例如：python -m http.server。

9. assert语句

在 Python 中，assert 语句用于断言某个条件为真。如果条件为假，则会引发 AssertionError 异常。

作用：

帮助开发者在代码中检查某些假设条件是否成立，通常用于调试和测试。

语法

```
assert 真值表达式 [, 错误信息]
```

说明

- 当真值表达式为False时，用错误信息创建一个 AssertionError 类型的错误，并使用raise引发错误，进入异常状态。
- 错误信息可以没有。

示例

```
# assert语句示例

def calculate_discount_price(price, discount):
    '''
    计算并返回折扣后的价格!
    :param price:  原价
    :param discount:  折扣 0~1 之间
    :return: 折扣后的价格
    '''
    assert 0 < discount <= 1, "折扣必须在 0 到 1 之间"
    return price * discount


print('__debug__:', __debug__)
print(calculate_discount_price(999, .8))  # 八折
print(calculate_discount_price(999, 1.2))  # 12折
```

上述程序在运行时，如果 discount 不在 0 ~ 1之间时会触发 AssertionError 类型的错误。此错误只在调试运行时出现。优化模式下等同于没有 assert 语句。

assert语句等同于

```
if __debug__:
  if not (真值表达式):
       raise AssertionError(错误信息)
```

10. 文件对象的迭代访问

文件对象

用open()函数，读取文本文件时，返回的是文件对象。

文件对象是可迭代对象，遍历文件对象每次可以得到一行的字符串（包括行结束符'\n'或'\r\n'）。

示例

文件: 10_numbers.txt 内容如下:

```
1000
2000
3000
4000
```

程序如下:

```
# 文件对象的迭代访问示例

def sum_numbers_from_file(path_name):
    '''
    从文件 path_name 中读取每一行的整数，并返回这些整数的和'''
    total_sum = 0
    fr = open(path_name)
    for line in fr:
        s = line.strip()
        total_sum += int(s)
    fr.close()
    return total_sum

result = sum_numbers_from_file('10_numbers.txt')
print('result:', result)
```

运行结果:

```
result: 10000
```

str.strip(chars=None) 方法

作用:

返回去掉字符串左右两端指定的字符chars,返回剩余的内容的字符串。

说明:

不给出实参，则默认去掉空白字符（' '、'\r'、'\n'、'\t '、'\v'、'\f'）

示例

```
>>> s1 = '  \nABC  \r\n'
>>> s1.strip()
'ABC'
>>> s2 = '####ABCD#######'
>>> s2.strip('#')
'ABCD'
```

11. with语句

作用

使用于对资源进行访问的场合。确保使用过程中不管是否发生异常，都会执行必要的“清理”操作，并释放资源。

- 例如文件使用后自动关闭，线程中锁的自动获取和释放等。

语法

```
with 上下文管理器表达式1 [as 变量1], 上下文管理器表达式1 [as 变量1], ...:
    语句块
```

说明

- 执行下文管理器表达式，用as中的变量绑定上表达式返回的对象。
- with语句并不改变异常的状态。

没有 with 语句时，使用 try 语句打开文件的示例

```
# try-finally语句关闭文件示例

def sum_numbers_from_file(path_name):
    '''
    从文件 path_name 中读取每一行的整数，并返回这些整数的和'''
    total_sum = 0
    fr = open(path_name)  # OSError
    try:
        for line in fr:
            s = line.strip()
            total_sum += int(s)  # ValueError
    finally:
        fr.close()
    return total_sum

result = sum_numbers_from_file('11_numbers.txt')
print('result:', result)
```

使用 with 语句类关闭文件示例：

```
# with 语句示例

def sum_numbers_from_file(path_name):
    '''
    从文件 path_name 中读取每一行的整数，并返回这些整数的和'''
    total_sum = 0
    # fr = open(path_name)  # OSError
    with open(path_name) as fr:
        for line in fr:
            s = line.strip()
            total_sum += int(s)  # ValueError
    return total_sum
```

```
result = sum_numbers_from_file('11_numbers.txt')
print('result:', result)
```

with 语句能确保关闭文件，但在出错进入异常时，with语句和 try-finally 语句一样不会改变异常状态。

12. with语句嵌套

with语句嵌套

```
with 上下文管理器表达式1 [as 变量1]:
    with 上下文管理器表达式2 [as 变量2]:
        语句块
```

语法

```
with 上下文管理器表达式1 [as 变量1], 上下文管理器表达式2 [as 变量2]:
        语句块
# 或
with (上下文管理器表达式1 [as 变量1], 上下文管理器表达式2 [as 变量2]):
        语句块
```

示例

```
# 此实例示意with语句嵌套时的特殊语法

def copy_text_file(src_pathname, dst_pathname):
    '''此函数将文本文件src_pathname中的内容复制到新文件 dst_pathname文件中'''
    # with open(dst_pathname, 'w') as fw:
    #     with open(src_pathname) as fr:
    with (open(dst_pathname, 'w') as fw, open(src_pathname) as fr):
            for line in fr:
                fw.write(line)

copy_text_file('test.txt', 'new.txt')
```

上述是使用with 语句打开两个文件并实现内容复制的例子。当任何一处出现错误并进入异常状态时，两个文件都会正常关闭。

练习

写程序，将一个UTF-8编码的文本文件'a.txt'，转化为GBK编码的'b.txt'文件。

参考答案

```
with (open('a.txt', encoding='utf-8') as fr,
      open('b.txt', 'w', encoding='gbk') as fw):
    for line in fr:
        fw.write(line)
```

13. 上下文管理器

能够用with语句进行管理的对象必须是上下文管理器。

内置的上下文管理器：

- 文件对象
- 线程锁
- 套接字 (socket)
- 子进程 (subprocess)
- 数据库连接 (sqlite3)
- 压缩文件 (gzip, zipfile)

什么是上下文管理器

类内实现了 `__enter__` 和 `__exit__` 实例方法的类称为上下文管理器类型，用此类型创建的对象称为上下文管理器。

with 语句在执行时，可以自动调用 `__enter__` 方法实现资源的申请，在离开with语句时可以自动的调用 `__exit__` 方法来释放资源，确保资源在使用后被释放。

定义语法

```
class 上下文管理器类名:
    '此类用于创建能够使用with语句进行管理的上下文管理器对象'
    def __enter__(self):
        '进入with语句时调用，返回资源对象'
        return 资源对象（通常是自己）

    def __exit__(self, exc_type, exc_value, exc_tb):
        '已离开with语句时调用'
```

__exit__方法的参数说明

- exc_type:绑定异常类型，没有异常时绑定None

- exc_value:绑定异常对象，没有异常时绑定None
- exc_tb:绑定traceBack对象类型，没有异常时绑定None
  - traceBack对象可以通过traceback模块的print_tb函数打印出引发异常的位置
  - 示例代码

```
import traceback as tb
tb.print_tb(exc_tb)
```

示例

```
# 此示例示意上下文管理器类型的定义方法

class MyContextManager:
    def __enter__(self):
        print('进入了__enter__方法')
        # 此处写申请资源的代码
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('退出了__exit__方法')
        print('exc_type:', exc_type)
        print('exc_val:', exc_val)
        print('exc_tb:', exc_tb)
        # 此处写释放资源的代码
        if exc_type is not None:
            import traceback as tb
            tb.print_tb(exc_tb)


print('开始进入with语句!')

with MyContextManager() as obj:
    print('with正在执行中')
    raise ValueError('myerror!')

print('with语句结束!')
```

上述自定义的上下文管理器类创建的对象会在进入 with 语句时调用 `__enter__` 方法。离开时调用 `__exit__` 方法，并且在离开时能够根据参数知道当前的异常状态等信息。

第二十一章、字节串和字节数组

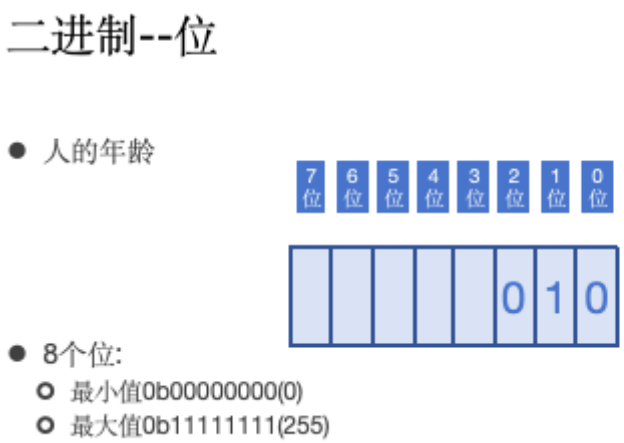
1. 字节串

进制装换 十进制的位是由 0~9 十个数字组成，每增加一个位，能表达的数字的范围增大十倍。

如图:

二进制的位是由 0和1 两个数字组成，每增加一个位，能表达的数字的范围增大二倍。

如图:



字节的概念

什么是字节

字节是由8个二进制的位组成，在计算机领域，通常作为保存信息的最小单位。

取值范围：

- 最小值:0
- 最大值:255

字节串

python中，用于保存字节信息的序列。

作用

存储以字节为单位的数据，如：文件、内存、网络通信等.

2. 字节串的创建

创建字节串的字面值

字节串字面值写法:用英文的b开头，后跟 ' 或 " 或 ''' 或 """ 开始或结束

示例:

```
b = b''      # 空字节串
b = b""      # 空字节串
b = b''''''  # 空字节串
b = b"""""" # 空字节串
b = b'hello world!' # 含有12个字节的字节串
```

字节串字面值中 0~127的数可以使用ASCII中的英文字符表示，也可以使用\x后跟两位十六进制的字符表示一个字节。

值大于等于128的字节必须用\x后跟两位十六进制的字符表示一个字节。

字节串中不能有中文。

示例

```
b'ABCD'     # 四个字节，值分别为:65、66、67、68
b'\x41\x42CD'  # 同上
b'\xFF\D8'  # 二个字节，值分别为:255、216
```

创建字节串的函数

构造函数bytes()

函数	说明
bytes()	生成一个空的字节串 等同于 b''
bytes(整数n)	生成n个值为0的字节串
bytes(整型可迭代对象)	用可迭代对象初始化一个字节串
bytes(字符串, encoding='utf-8')	用字符串的转换编码生成一个字节串

示例

```
>>> bytes()
b''
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
>>> bytes([65,66,67,68])
b'ABCD'
>>> bytes('ABCD中文', encoding='utf-8')
b'ABCD\xe4\xb8\xad\xe6\x96\x87'
```

3. 字节串的运算

字节串也是序列，也字符串和列表一样，支持通用序列操作，操作方式如下表所示。

运算	结果：
x in s	如果 s 中的某项等于 x 则结果为 True，否则为 False
x not in s	如果 s 中的某项等于 x 则结果为 False，否则为 True
s + t	s 与 t 相拼接
s * n 或 n * s	相当于 s 与自身进行 n 次拼接
s[i]	s 的第 i 项，起始为 0
s[i:j]	s 从 i 到 j 的切片
s[i:j:k]	s 从 i 到 j 步长为 k 的切片
len(s)	s 的长度
min(s)	s 的最小项
max(s)	s 的最大项
s.index(x[, i[, j]])	x 在 s 中首次出现项的索引号（索引号在 i 或其后且在 j 之前）
s.count(x)	x 在 s 中出现的总次数

示例

```
>>> b = b'ABC'
>>> 65 in b
True
>>> 88 not in b
True
>>> b'ABC' + b'123'
b'ABC123'
>>> b'ABC' * 3
b'ABCABCABC'
>>> b[0]
65
>>> b[0:2]
b'AB'
>>> b[::2]
b'AC'
>>> len(b)
3
>>> min(b)
65
>>> max(b)
67
>>> b.index(66)
1
```

```
>>> b.count(66)
1
```

4. 字节串和字符串

字节串（bytes）和字符串（str）区别

bytes存储字节。

- 取值范围0~255。

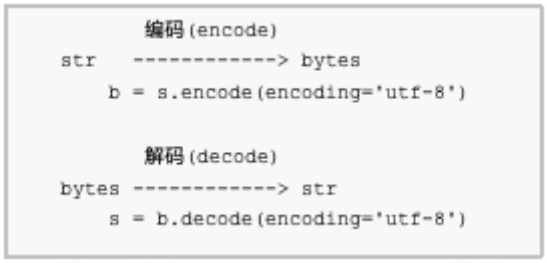
str存储unicode字符。

- 取值范围0~1114111（0~0x10FFFF）。
- 使用ord()函数可以返回这个值。

字节串和字符串转换

字符串可以编码成为字节串，同时也可以再解码回字符串。

转换方法，如图:



编码、解码函数

函数	说明
str.encode(encoding='utf-8')	返回编码为 bytes 的字符串。
bytes.decode(encoding='utf-8')	返回解码为 str 的字节串。

编码格式encoding

encoding参数的取值如下：

```
'utf-8'、 'gb2312'、'gbk' 或 'gb18030'
```

汉字编码

- 国际编码（UNICODE）
  - 'utf-8'。
- 信息产业部国标（GB）系列编码
  - 'gb2312' -- 6000+个字；
  - 'gbk' -- 20013个汉字；
  - 'gb18030'-- 27000+个汉字。

示例

```
>>> s = '中国'
>>> b = s.encode()
>>> b
b'\xe4\xb8\xad\xe5\x9b\xbd'
>>> b.decode()
'中国'
>>> b2 = s.encode('gbk')
>>> b2
b'\xd6\xd0\xb9\xfa'
>>> b2.decode('gbk')
'中国'
```

5. 字节数组

什么是字节数组

字节数组是可变的字节串。

- 字节串是不可变的二进制序列类型。
- 字节数组是可变的二进制序列类型。

字节数组的创建

bytearray 对象没有专属的字面值语法，它们总是通过调用构造器函数来创建。

构造函数bytearray()。

函数	说明
bytearray()	创建一个空的字节数组
bytearray(整数n)	创建一个指定长度的以零值填充的字节数组
bytearray(整型可迭代对象)	通过由整数组成的可迭代对象创建一个字节数组
bytearray(字符串, encoding='utf-8')	用字符串的转换编码生成一个字节数组

示例:

```
>>> bytearray()
bytearray(b'')
>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')
>>> bytearray([65, 66, 67])
bytearray(b'ABC')
>>> bytearray("中国", 'utf-8')
bytearray(b'\xe4\xb8\xad\xe5\x9b\xbd')
```

字节数组和字节串一样，支持通用序列操作，操作方式如下表所示。

通用序列操作:

运算	结果：
x in s	如果 s 中的某项等于 x 则结果为 True，否则为 False
x not in s	如果 s 中的某项等于 x 则结果为 False，否则为 True
s + t	s 与 t 相拼接
s * n 或 n * s	相当于 s 与自身进行 n 次拼接
s[i]	s 的第 i 项，起始为 0
s[i:j]	s 从 i 到 j 的切片
s[i:j:k]	s 从 i 到 j 步长为 k 的切片
len(s)	s 的长度
min(s)	s 的最小项
max(s)	s 的最大项
s.index(x[, i[, j]])	x 在 s 中首次出现项的索引号（索引号在 i 或其后且在 j 之前）
s.count(x)	x 在 s 中出现的总次数

示例

```
>>> ba = bytearray([65, 66, 67])
>>> 67 in ba
True
>>> ba + ba
bytearray(b'ABCABC')
>>> ba[::2]
bytearray(b'AC')
>>>
```

6. 字节数组的可变序列操作

为什么要用字节数组

1. 需要进行字节数据处理，同时需要修改字节数据。
2. 字节数组可以替代所有适用于字节串的场合。

可变序列操作

运算	结果
s[i] = x	将 s 的第 i 项替换为 x
s[i:j] = t	将 s 从 i 到 j 的切片替换为可迭代对象 t 的内容
del s[i:j]	等同于 s[i:j] = []
s[i:j:k] = t	将 s[i:j:k] 的元素替换为 t 的元素
del s[i:j:k]	从列表中移除 s[i:j:k] 的元素
s.append(x)	将 x 添加到序列的末尾 (等同于 s[len(s):len(s)] = [x])
s.clear()	从 s 中移除所有项 (等同于 del s[:])
s.copy()	创建 s 的浅拷贝 (等同于 s[:])
s.extend(t) 或 s += t	用 t 的内容扩展 s (基本上等同于 s[len(s):len(s)] = t)
s *= n	使用 s 的内容重复 n 次来对其进行更新
s.insert(i, x)	在由 i 给出的索引位置将 x 插入 s (等同于 s[i:i] = [x])
s.pop() 或 s.pop(i)	提取在 i 位置上的项，并将其从 s 中移除
s.remove(x)	从 s 中移除第一个 s[i] 等于 x 的条目
s.reverse()	就地将列表中的元素逆序。

示例

```
>>> ba = bytearray(b'ABCdEFG')
>>> ba[3] = 68
>>> ba
bytearray(b'ABCDEFG')
>>> del ba[3]
>>> ba
bytearray(b'ABCEFG')
>>> ba.clear()
>>> ba
bytearray(b'')
```

7. 字节串和字节数组的方法

字节串和字节数组有着共同的方法名和功能。

详见官方文档

https://docs.python.org/zh-cn/3/ -> 库参考 -> 二进制序列类型。

在目录里找 “bytes 和 bytearray 操作”。

https://docs.python.org/zh-cn/3/library/stdtypes.html#bytes-and-bytearray-operations

第二十二章、文件（高级）

1. 二进制文件

文件都是以字节为单位进行存储的。

文件操作的两种方式：

1. 文本文件
   - 例如：.py、.c、.cpp、.txt等文件。
   - 用字符串进行操作，写入时自动编码为字节串，读取时自动解码为字符串。

2. 二进制文件
   - 例如：.mp3、.mp4、.zip、.exe等文件。
   - 直接用字节串或字节数组进行操作，文本数据需要手动编解码。

打开文件

文件只有打开才能进行操作。

打开文件的函数：

函数	说明
open(file, mode='r', encoding=None, newline=None)	打开文件并返回对应的文件对象file，如果该文件不能被打开，则引发 OSError类型的错误。

参数

- file：文件路径名。
- mode：打开模式，默认是’r’读取文件，’w’是创建新文件并写入文件。
- encoding：文本文件的编码，中文是’utf-8’或’gb2312’/’gbk’/’gb18030’。
- newline：换行符号，windows是CRLF(’\r\n’),mac和Linux是LR(’\n’)。

文件的打开模式mode:

字符	含意
'r'	读取（默认）
'w'	写入，并先清空文件，没有文件则尝试创建文件
'x'	排它性创建，如果文件已存在则失败
'a'	打开文件用于写入，如果文件存在则在末尾追加
'b'	二进制模式
't'	文本模式（默认）
'+'	打开用于更新（读取与写入）

二进制文件的打开模式mode

打开模式示例

字符	含意
'rb'	二进制读取
'wb'	二进制写入，并先清空文件，没有文件则尝试创建文件。
'w+b'	可以实现二进制随机读写，当打开文件时，文件内容将被清零
'r+b'	以二进制读和更新模式打开文件,打开文件时不会清空文件内容
'r+'	以文本模式读和更新模式打开文件,打开文件时不会清空文件内容

读文件的方法

读文件

方法	说明
file.read(size=-1)	从一个文件流中最多读取size个字符或字节，如果不给出参数，则默认读取文件中全部的内容并返回。
file.readline()	读取一行数据, 如果到达文件尾则返回空行
file.readlines(max_chars=-1)	返回每行字符串或字节串的列表,max_chars为最大字符数或字节数

写文件的方法

写文件

方法	说明
file.write(text)	写一个字符串到文件流中，返回写入的字符或字节数
file.writelines(lines)	将字符串的列表或字节串的列表中的内容写入文件

示例

文件：`a.txt` 的内容如下

```
abc中文
this is second line
```

二进制文件读取示例代码如下:

```
# 二进制文件读取示例

# 以文本方式读取
try:
    with open('a.txt', 'rt') as fr:
        s = fr.read()
        print('s:', s)
        print('字符串的长度', len(s))
except OSError:
    print('文件: a.txt 打开失败!')

# 以二进制方式读取
try:
    with open('a.txt', 'rb') as fr:
        b = fr.read()
        print('b:', b)
        print('字节串的长度', len(b))
except OSError:
    print('文件: a.txt 打开失败!')
```

文件：`b.txt` 的内容如下

```
abcde
ABCDE
123###xxx
```

二进制文件写入示例代码如下:

```
# 二进制文件写入示例

# 以二进制方式写入
try:
    with open('b.txt', 'wb') as fw:
        nbytes = fw.write(b'abcde\n')
        print('成功写入', nbytes, '字节!')
        nbytes = fw.write(bytearray(b'ABCDE\n'))
        print('成功写入', nbytes, '字节!')
        fw.writelines([b'123', b'###', bytearray(b'xxx')])
except OSError:
    print('文件: b.txt 打开失败!')
```

2. 文件的随机读写

什么是文件的随机读写

文件的随机读写是指能够在文件的任意位置直接读取或写入数据，而无需按顺序从头到尾遍历内容。

随机读写通过移动文件的读写位置来快速定位到指定位置，实现高效的数据操作。

应用场景

1. 需改部分内容。
2. 高效查询。
3. 二进制文件处理。
4. 日志追加。

随机读写的方法

方法	说明
file.tell()	返回当前读写位置（相对于文件头）
file.seek(offset, whence=0)	设置文件的读写位置

file.seek()的参数

- offset-偏移量
  - 大于0的数代表向文件末尾方向移动。
  - 小于0的数代表向文件头方向移动。

- whence- 相对于哪里
  - 0：代表从文件头开始偏移。
  - 1：代表从当前读写位置开始偏移。
  - 2：代表从文件尾开始偏移。

示例：

文件 `b.txt` 内容如下:

```
>>> f = open('b.txt', 'rb')
>>> f.tell()
0
>>> f.read(3)
b'abc'
>>> f.tell()
3
>>> f.read()
b'de\nABCDE\n123###xxx'
>>> f.tell()
21
>>> f.seek(2, 0)
2
>>> f.read(3)
b'cde'
>>> f.tell()
5
```

文件对象的其他方法

方法	说明
file.flush()	把写入文件对象的缓存内容写入到磁盘
file.readable()	判断这个文件是否可读,可读返回True,否则返回False
file.writeable()	判断这个文件是否可写,可写返回True,否则返回False
file.seekable()	返回这个文件对象是否支持随机定位
file.truncate(pos = None)	剪掉自pos位置之后的数据，返回新的文件长度

3. 标准输入输出文件

标准输入输出是操作系统提供的标准数据流。

Mac/Linux/Windows系统都有的三个标准输入输出：

- stdin - 标准输入；
- stdout - 标准输出；
- stderr - 标准错误输出。

模块名：sys

文件对象	说明
sys.stdin	标准输入文件，默认是键盘；ctrl+d键结束输入
sys.stdout	标准输出，默认是屏幕终端，可以重定向到文件
sys.stderr	标准错误输出，默认是屏幕终端，可以重定向到文件

示例

```
>>> import sys
>>> n = sys.stdout.write("hello world\n")
hello world
>>> n
12
>>> s = sys.stdin.readline()
abcdefg
>>> s
'abcdefg\n'
```

第二十三章、迭代器和生成器

1. 迭代器

可迭代对象（iterable）

可迭代对象是指能够依次获取数据元素的对象，是可以使用for语句遍历的对象。

如：

```
lst = [100, 200, 300]
```

可迭代对象是指能用iter(iterable)函数返回迭代器的对象。

什么是迭代器

- 迭代器是访问可迭代对象的工具。
- 迭代器是指用 iter(iterable) 函数返回的对象。
- 迭代器可以用next(iterator)函数获取可迭代对象的数据。

作用

遍历可迭代对象，为同时访问多个可迭代对象提供方法。

示例

```
>>> lst = [100, 200, 300]
>>> it = iter(lst)  # it 绑定可以访问列表的迭代器
>>> it
<list_iterator object at 0x1064c92d0>
>>> next(it)  # 获取列表的第一个值
100
>>> next(it)  # 获取列表的第二个值
200
```

迭代器相关的函数

函数	说明
iter(iterable)	从可迭代对象（iterable）中返回一个迭代器（iterator），iterable必须是能提供一个迭代器的对象
next(iterator)	从迭代器iterator中获取下一个记录，如果无法获取一下条记录，则引发 StopIteration 异常

2. 迭代器协议

什么是迭代器协议

迭代器协议需要满足如下两条：

- 所有的可迭代对象能够使用iter(iterable)函数来获取迭代器。
- 迭代器能够使用next(iterator)函数获取下一项数据，在没有下一项数据时引发一个 StopIteration 异常来终止迭代的约定。

> 能够使用for语句遍历的可迭代对象必须遵守迭代器协议。

迭代器协议示例

```
title = ('No1', 'No2', 'No3', 'No4')
primes = [2, 3, 5, 7]  # 质数

# 访问列表中的所有元素
# 1. 使用 for 语句遍历
# for x in primes:
#     print(x)
# else:
#     print('遍历结束')

# 2. 使用 迭代器遍历
it1 = iter(primes)
it2 = iter(title)
while True:
    try:
        x = next(it1)
        n = next(it2)
        print(n, x)
    except StopIteration:
        print('遍历结束!')
        break
```

上述列表 primes 是一个可迭代对象，它可以用 for 语句取值，也可以先用 iter(x) 函数返回迭代器，再用next(it) 函数向迭代去取值，知道引发 StopIteration 异常时，说明取值结束。

3. 生成器

什么是生成器

- 生成器是能够动态提供数据的可迭代对象。
- 生成器是在程序运行时生成数据，与容器类不同，它通常不会在内存中保存大量的数据，而是现用现生成。

例如:

range() 函数返回的对象就是整数序列生成器。

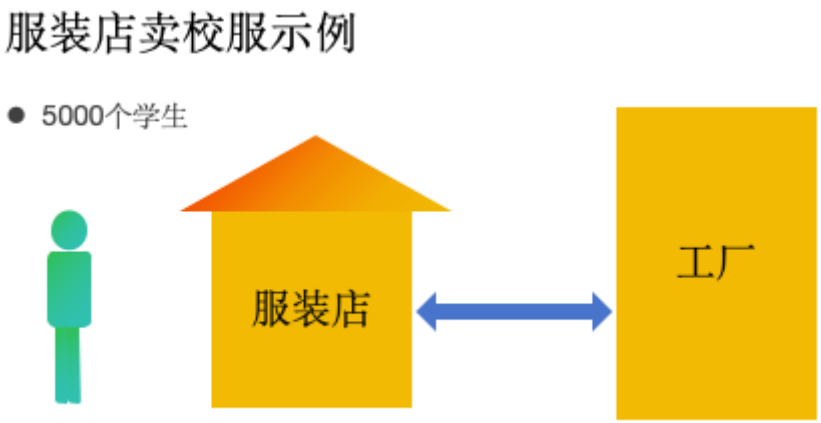
白话文解释

我在学校门口开了一家服装店，今年有5000个新生入校，为了多买提供校服，我有两种做法：

1. 从工厂进货 5000 套校服，放在仓库准备出售。
2. 和工厂签约，当有学生来卖校服时，立刻下单，让工厂来制作，制作完成后送到店里交给客户。

而第二种做法就是生成器的原理，现用现生成。

如图



生成器的特点和作用

作用

1. 用于按需生成值，而不是一次性生成所有值。
2. 适合处理大数据集或无限序列。

特点

1. 惰性求值：值在需要时才生成，节省内存。
2. 有时不能重复遍历：生成器一般只能遍历一次，再次遍历需要重新创建。
    - 具体要看实现方法

python创建生成器的方式有三种：

1. 生成器函数。
2. 生成器表达式。
3. 生成器类。

示例:

```
# 生成器示例
#
# numbers = list(range(1000000000000000))
#
# for x in numbers:
#     print(x)

for x in range(10000000000000000000000000000000):
    print(x)
```

上述程序中，range() 是类，它创建的对象就是生成器对象。当向 生成器对象取值时，才会生成需要的值，否则不会生成。

4. 生成器函数

什么是生成器

生成器是能够动态提供数据的可迭代对象。

特点：

现用现生成，节省内存。

python创建生成器的方式有三种：

1. 生成器函数
2. 生成器表达式
3. 生成器类

生成器函数

什么是生成器函数

含有yield语句的函数是生成器函数，此函数被调用将返回一个生成器对象。

> 生成器对象是一个可迭代对象。

yiled语句

作用

yield 语句仅在定义生成器函数的内部使用，使得此函数不再是普通函数，而是生成器函数。 生成器函数被调用将返回一个生成器对象。

语法

```
yield 表达式
# 或
yield from 表达式（此表达式必须返回一个可迭代对象或生成器）
```

生成器函数说明

- 生成器函数的调用将返回一个生成器对象，生成器对象是一个可迭代对象。
- 生成器对象使用 yield 语句逐步生成值，而不是一次性返回所有结果。
- 生成器函数在每次调用 yield 语句时暂停执行，并在下次请求值时从暂停处继续运行。
- 在生成器函数调用return会触发一个 StopIteration 异常（即生成数据结束）。

示例

```
# 生成器函数示例

def my_generator():
    print('生成器函数开始:')
    yield 1
    yield 2
    yield 5
    print('生成器函数结束!')

generator_obj = my_generator()  # generator_obj 绑定生成器对象

print(generator_obj)
```

```
for x in generator_obj:
    print('x:', x)
```

上述函数 my_generator() 调用后，返回的是生成器对象，可以使用迭代器或 for 语句取值。

生成器函数调用，函数内部的语句并不会执行，只有向生成器对象创建的迭代器取值时，内部的语句才会递进运行，当遇到 yield 语句就会停止执行并将值返回 给next(it) 函数。下一次调用 next(it) 时在从上次的yield 语句处继续执行。

生成器函数执行完毕会引发 StopIteration 异常通知。

yield from 示例

```
# yield from 语句示例

def my_generator():
    yield 1
    yield from [2, 3, 4, 5]
    yield 6

for x in my_generator():
    print(x)
```

5. 生成器函数示例

range函数

作用

调用后，返回一个能够得到一系列整数的可迭代对象。

调用格式

```
range(stop)               # stop 停止整数
range(start, stop)        # start 开始整数
range(start, stop, step)  # step 步长
```

说明

- 省略 step 参数，则默认为 1。
- 省略 start 参数，则默认为 0。

生成器函数示例

仿照range()函数，写一个生成器函数my_range，替代 range函数的功能。

```
def my_range(start, stop=None, step=None):
    ... # 实现此处的代码


print(list(my_range(5)))  # [0, 1, 2, 3, 4]
print(list(my_range(5, 10)))  # [5, 6, 7, 8, 9]
print(list(my_range(1, 10, 2)))  # [1, 3, 5, 7, 9]
```

实现方式

```
# 生成器函数示例

# 仿照range()函数，写一个生成器函数my_range，替代 range函数的功能

def my_range(start, stop=None, step=None):
    # 1. 校正参数，得到开始值，结束值和步长
    if stop is None:
        stop = start
        start = 0
    if step is None:
        step = 1
    # 2. 判断开始值，结束值和步长是否都是整数，如果不是整数，则抛出TypeError异常
    if type(start) is not int:
        raise TypeError('start 不是整数!')
    if not isinstance(stop, int):
        raise TypeError('stop 不是整数!')
    if not isinstance(step, int):
        raise TypeError('step 不是整数!')
    # 3. 生成整数并用yield送回个迭代器的next()函数
    if step > 0:
        cur_value = start
        while cur_value < stop:
            yield cur_value
            cur_value += step
    elif step < 0:
        cur_value = start
        while cur_value > stop:
            yield cur_value
            cur_value += step

print(list(my_range(5)))  # [0, 1, 2, 3, 4]
print(list(my_range(5, 10)))  # [5, 6, 7, 8, 9]
print(list(my_range(1, 10, 2)))  # [1, 3, 5, 7, 9]
```

练习

写一个生成器函数 fibonacci_generator(n) 生成前n个斐波那契数列。

- 斐波那契数列的第一个数是0，第二个数是1，后续每一项都是前两项的和。

- 数列的前几项为：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

填写代码:

```
def fibonacci_generator(n):
    ...  # 填写次数的代码

fibonacci_list = list(fibonacci_generator(20))
print('前20个斐波那契数列:', fibonacci_list)
```

6. 生成器表达式

作用:

用推导式形式创建一个新的生成器。

语法

```
( 表达式1
    for 变量1 in 可迭代对象1 [if 真值表达式1]
        [for 变量2 in 可迭代对象2 [if 真值表达式2]]* )
```

说明

- 中括号内部的子句可以省略。
- `*` 代表子表达式可以有0个、1个或多个。

示例

```
# 生成器表达式示例

# 创建一个 0 ~ 1000000000000000000 这些数的平方的生成器，
# 并用print函数打印这些数。

for x in ( n ** 2 for n in range(0, 1000000000000000000)):
    print(x)
```

练习

写一个函数 is_prime(x) 判断x是否是素数，如果是素数返回True，否则返回False。

用is_prime函数，结合生成器表达式，创建一个能生成100以内所有素数的生成器，并打印出结果。

参考答案:

```
# 练习：生成器表达式-参考答案

def is_prime(x):
    if x <= 1:
        return False
    for i in range(2, x):
        if x % i == 0:
            return False
    return True

for prime in (n for n in range(100) if is_prime(n)):
    print(prime)
```

7. 生成器类型

什么是可迭代对象

可迭代对象是指能用iter(iterable)函数返回迭代器的对象。

可迭代对象内部要定义__iter__(self)方法来返回迭代器对象。

- iter(iterable) 函数调用的就是对象的obj.**iter**()方法。

示例

```
class MyIterable:
    def __iter__(self):
        语句块
        return 迭代器
```

什么是迭代器

是指能用next(iterator)函数取值的对象。

迭代器对象内部要定义 `__next__(self)`方法来返回当前取值，在没有值时需要引发StopIteration异常来终止迭代，即实现迭代器协议。

- next(iterator) 函数调用的就是对象的 `obj.__next__()` 方法。

示例

```
class MyIterator:
    def __next__(self):
```

```
        迭代器协议
        return 数据
```

什么是生成器类

类内实现了 `__iter__()` 和 `__next__()` 方法，并在 `__next__` 方法内实现了迭代器协议，该类创建的对象够动态生成数据，这个类叫做生成器类。

> python内置的range、enumerate和zip类就是生成器类。

生成器类创建的对象是可迭代对象，也是生成器对象。

生成器类的大致结构如下所示：

```
class MyGenerator:
    def __iter__(self):
        语句块
        return 迭代器
    def __next__(self):
        迭代器协议
        return 数据
```

生成器函数示例

仿照range()函数，写一个生成器类MyRange，替代 range函数的功能。

```
class MyRange:
    def __init__(self, start, stop=None, step=None):
    def __iter__(self):
    def __next__(self):

print(list(MyRange(5)))  # [0, 1, 2, 3, 4]
print(list(MyRange(5, 10)))  # [5, 6, 7, 8, 9]
print(list(MyRange(1, 10, 2)))  # [1, 3, 5, 7, 9]
```

实现代码

```
# 生成器类示例

# 仿照range()函数，写一个生成器类MyRange，替代 range函数的功能
class MyRange:
    def __init__(self, start, stop=None, step=None):
        # 1. 校正参数，得到开始值，结束值和步长
        if stop is None:
            stop = start
            start = 0
```

```
        if step is None:
            step = 1
        # 2. 判断开始值，结束值和步长是否都是整数，如果不是整数，则抛出TypeError异常
        if type(start) is not int:
            raise TypeError('start 不是整数!')
        if not isinstance(stop, int):
            raise TypeError('stop 不是整数!')
        if not isinstance(step, int):
            raise TypeError('step 不是整数!')
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        self.cur_value = self.start  # 初始化当前值
        return self  # 把自己当成迭代器返回

    def __next__(self):
        '''
        实现迭代器协议：
        生成整数并用送回给迭代器的next()函数，如果越界则引发StopIteration异常
        '''
        value = self.cur_value
        if self.step > 0:
            if self.cur_value >= self.stop:
                raise StopIteration
        if self.step < 0:
            if self.cur_value <= self.stop:
                raise StopIteration
        self.cur_value += self.step
        return value

print(list(MyRange(5)))  # [0, 1, 2, 3, 4]
print(list(MyRange(5, 10)))  # [5, 6, 7, 8, 9]
print(list(MyRange(1, 10, 2)))  # [1, 3, 5, 7, 9]
print(list(MyRange(10, 0, -2)))  # [10, 8, 6, 4, 2]
```

练习

写一个生成器类：Fibonacci 生成前n个斐波那契数列。

- 斐波那契数列的第一个数是0，第二个数是1，后续每一项都是前两项的和。
- 数列的前几项为：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

填写代码:

```
class Fibonacci:
    def __init__(self, n): ...  # 填写此方法的代码
    def __iter__(self):...  # 填写此方法的代码
    def __next__(self): ...  # 填写此方法的代码
```

```
fibonacci_list = list(Fibonacci(20))
print('前20个斐波那契数列:', fibonacci_list)
```

8. 迭代工具函数

作用：

- 对一个或多个可迭代对象封装成生成器，方便迭代访问。
- 用于高效处理可迭代对象。

构造函数

函数	说明
zip(*iterables)	在多个迭代器上并行迭代，从每个迭代器返回一个数据项组成元组。生成元组的个数由最小的可迭代对象大小决定
enumerate(iterable, start=0)	返回一个元组，里面包含一个计数值（从 start 开始，默认为 0）和通过迭代 iterable 获得的值。

zip示例

```
# zip 函数示例

# 函数: zip(*iterables)
# 说明: 在多个迭代器上并行迭代，从每个迭代器返回一个数据项组成元组。
#       生成元组的个数由最小的可迭代对象大小决定

numbers = [10086, 10000, 10010, 95588]
names = ["中国移动", "中国电信", "中国联通"]
# 用zip函数，每次迭代出元组，格式为(号码,名字)
for t in zip(numbers, names):
    print(t)

for t in zip(range(1, 10000), numbers, names):
    print(t)   # (1, 10086, '中国移动')

for n, num, nam in zip(range(100), numbers, names):
    print(n, num, nam)
```

enumerate 示例

```
# enumerate 函数示例

# 函数: enumerate(iterable, start=0)
# 说明: 返回一个元组，里面包含一个计数值（从 start 开始，默认为 0）
#       和通过迭代 iterable 获得的值。

names = ["中国移动", "中国电信", '中国联通']

# 要求: 为上述名称加一个序号，
# 用enumerate函数，每次迭代出元组，格式为(序号, 名字)
for t in enumerate(names, start=1):
    print(t)   # (1, '中国移动')
```

enumerate函数的实现原理

enumerate函数等价于：

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

练习

写一个程序，读入任意行的文字，当输入空行时结束输入。

打印带有行号的输入结果。

如:

```
$ python3 mytest.py
请输入: hello<回车>
请输入: world<回车>
请输入: bye<回车>
请输入: <回车>
输出如下:
第1行: hello
第2行: world
第3行: bye
```

第二十四章、函数式编程

1. 函数式编程

什么是编程范式

编程范式是一种编程的方法论或思想体系，它定义了程序员如何组织代码、解决问题，并影响语言的语法和设计风格。不同的范式提供不同的抽象方式和代码结构，适用于不同类型的任务。

常见的编程范式：

- 面向过程编程（Procedural Programming, POP）；
- 面向对象编程（Object-Oriented Programming, OOP）；
- 函数式编程（Functional Programming, FP）。

面向过程编程（POP）

- 核心：以步骤为中心，分解任务为函数和过程。
- 特点：
  - 数据与逻辑分离：数据通过函数参数传递。
  - 代码组织：线性流程（如 函数A → 函数B → 函数C ）。
- 优点：
  - 简单直观，适合小型任务。
  - 执行效率高。
- 缺点：
  - 代码复用性差，维护困难（大型项目易混乱）。
- 数据全局共享易产生副作用。

面向过程编程-示例

```
# 面向过程编程（Procedural Programming, POP）

zhangsan = {'name':'张三', 'score': 0}

def set_score(stu, new_score):
    stu['score'] = new_score
```

```
def get_score(stu):
    return stu['score']
```

面向对象编程（OOP）

- 核心：以对象为中心，封装数据与方法。
- 区别：
  - 三大特性：封装、继承、多态。
  - 代码组织：通过类定义对象交互（如 dog1.eat('骨头')）。
- 优点：
  - 模块化强，适合复杂系统（如 GUI、游戏）。
  - 代码复用性高（继承/组合）。
- 缺点：
  - 学习曲线陡峭（设计模式复杂）。
  - 性能略低于 POP（对象创建开销）。

面向对象编程-示例

```
# 面向对象编程（Object-Oriented Programming, OOP）

class Student:
    def __init__(self, name, score=0):
        self.name, self.score = name, score
    def set_score(self, new_score):
        self.score = new_score
    def get_score(self):
        return self.score

zhangsan = Student('张三')
zhangsan.set_score(100)
print(zhangsan.get_score())
```

函数式编程（FP）

使用函数解决问题。

- 核心：以纯函数为中心，避免状态与副作用。
- 区别：
  - 无状态：数据不可变，函数输出仅依赖输入。
  - 高阶函数：函数可作为参数或返回值（如 map/reduce）。
- 优点：
  - 线程安全，天然适合并发（如大数据处理）。

- 代码简洁，易于测试（纯函数无副作用）。

- 缺点：
  - 学习成本高（递归等概念）。
  - 性能问题（大量数据复制）。

函数式编程-示例

函数式编程

打印 1+2+3+...+100的和。

```
print(sum(range(1, 101)))
```

2. 纯函数

什么是纯函数?

纯函数是指满足以下两个核心条件的函数：

- 相同输入始终会用相同输出
  - 只要输入参数相同，无论调用多少次，返回值必须完全一致。
  - 例如：math.sqrt(4) 永远返回 2，不依赖外部状态。

- 无副作用
  - 不会修改函数外部的任何状态（如全局变量、输入参数、文件系统等）。
  - 例如：不会修改全局变量、不发起网络请求、不写入数据库。

纯函数-示例

```
# 纯函数示例
def pure_add(a, b):
    '''这个是纯函数'''
    return a + b

# 非纯函数示例
total_sum = []
def not_pure_add(a, b):
    '''这个是不纯函数，他修改了外部的状态'''
    total_sum.append(a + b)
    return a + b
```

3. 递归函数

什么是递归函数

递归函数是指在函数内部直接或间接调用自身的函数。

递归函数通过将问题分解为更小的同类子问题来解决问题，直到达到一个终止条件，从而避免循环和当前数据压栈、弹栈等操作。

作用

简化特殊场合的编程难度，如：遍历二叉树、快速排序等场景。

递归示意

阶乘：n! = 1 * 2 * 3 * ... * n

求5!

```
5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1
```

递归求阶乘

```
def factorial(n):
    '使用递归计算阶乘!'
    if n == 1:
        return 1
    result = n * factorial(n - 1)
    return result

print("3! = ", factorial(3))
```

递归函数-原理

- 当一个函数A调用另外一个函数B时，A正在运行的环境（局部变量）会保存在栈空间中，直到B函数返回为止。
- 当一个函数A调用另外一个函数A时（自己时）也是如此。

递归的优缺点

优点

- 代码简洁：适合解决分治问题（如快速排序、树的遍历等操作）。
- 更符合数学定义：如阶乘、斐波那契数列的数学描述直接对应递归代码。

缺点

- 栈溢出风险：深层递归可能导致调用栈溢出（python默认调用深度1000层）。
- 性能问题：可能存在重复计算导致性能下降。

4. 高阶函数

什么是高阶函数（High Order Function）？

高阶函数是指满足以下任一条件的函数：

1. 能够接受一个或多个函数作为参数。
2. 能够返回一个函数作为结果。

高阶函数将函数视为“一等公民”，可以像普通变量一样传递和操作，这是函数式编程的核心特性之一。

函数变量

python 的标识符都是变量（关键字除外），他绑定一个对象。 python 中，类、函数、生成器、模块等都是对象。

函数名是一个变量，他绑定的是 一个函数（对象）。

示例

```
p = print
p("hello world!")
```

高阶函数特征

1. 一个函数可以作为另一个函数的实参传递：

如:

```
# 一个函数可以作为另一个函数的实参传递：

def myfunc(data, func):
    return func(data)

lst = [80, 100, 20, 60, 40]
```

```
value = myfunc(lst, max)
print('value:', value)  # 100

value = myfunc(lst, sum)  # 300
print('value:', value)
```

1. 函数可以作为另一个函数的返回值。

如:

```
# 函数可以作为另一个函数的返回值。

def get_function(operation):
    if operation == '求最大':
        return max
    elif  operation == '求最小':
        return min
    elif  operation == '求和':
        return sum
    else:
        raise ValueError('操作不存在')

lst = [80, 100, 20, 60, 40]

func = get_function('求和')
print(func(lst))

func = get_function('求最大')
print(func(lst))
```

练习

看懂下面程序在做什么:

```
def fx(f, x, y):
    print(f(x,y))

fx((lambda a, b: a + b), 100, 200)
fx(lambda a, b: a**b, 3, 4)
```

5. 内置高阶函数

函数	说明
map(function, iterable, *iterables)	返回一个将 function 应用于 iterable 的每一项，并产生其结果的迭代器。 如果传入了额外的 iterables 参数，则 function 必须接受相同个数的参数并被用于到从所有可迭代对象中并行获取的项。 当有多个可迭代对象时，当最短的可迭代对象耗尽则整个迭代将会停止。(映射)
filter(function, iterable)	使用 iterable 中 function 返回真值的元素构造一个迭代器。 iterable 可以是一个序列，一个支持迭代的容器或者一个迭代器。 如果 function 为 None，则会使用标识号函数，也就是说，iterable 中所有具有假值的元素都将被移除。
sorted(iterable, key=None, reverse=False)	根据 iterable 中的项返回一个新的已排序列表。

map 示例

```
# map函数示例

def power2(x):
    return x ** 2

# 1, 4, 9, 16, ... 81
# for x in map(power2, range(1,10)):
#     print(x)

def join_str(a, b):
    return str(a) + str(b)

for x in map(join_str, "ABCDEFG", [1, 2, 3, 4]):
    print(x)
```

filter 示例

```
# filter函数示例

def is_prime(x):
    if x <= 1:
        return False
```

```
    for i in range(2, x):
        if x % i == 0:
            return False
    return True

for x in filter(is_prime, range(10)):
    print(x)
```

map函数的实现原理

```
# map函数的实现原理

def mymap(function, iterable, *args):
    iterlist = [iter(iterable)]
    iterlist.extend((iter(it) for it in args))
    while True:
        arg_list = []
        for it in iterlist:
            try:
                value = next(it)
                arg_list.append(value)
            except StopIteration:
                return
        yield function(*arg_list)


for x in mymap(lambda x:x**2, [1, 2, 3, 4]):
    print(x)

for x in mymap(lambda x, y:x**y, [1, 2, 3, 4],[4, 3, 2, 1, 0]):
    print(x)
```

filter函数的实现原理

```
# filter函数的实现原理

def myfilter(function, iterable=None):
    for value in iterable:
        if function(value) if function else value:
            yield value

result = list(myfilter(None, [1, 0, 2.0, 0.0, True, False, None]))
print(result)
result = list(myfilter(lambda x: x%2, range(1, 10)))
print(result)
```

练习

求100以内所有素数的和，即求：2 + 3 + 5 + 7 + ... + 97 = ?

参考答案

```
def is_prime(x):
    if x <= 1:
        return False
    for i in range(2, x):
        if x % i == 0:
            return False
    return True

print(sum(filter(is_prime, range(100))))
```

6. 函数式编程模块functools

functools 模块应用于高阶函数。

函数	说明
functools.reduce(function, iterable, [initial,])	将两个参数的 function 从左至右累积地应用到 iterable 的条目，以便将该可迭代对象缩减为单个值。(规约)
functools.partial(func, *args*, *keywords)	返回一个新的 部分对象，此对象可以固定部分参数，调用部分对象才开始计算。

functools.reduce函数

作用：

将两个参数的 function 从左至右累积地应用到 iterable 的条目，以便将该可迭代对象缩减为单个值（规约）。

调用格式：

```
functools.reduce(function, iterable, [initial, ])
```

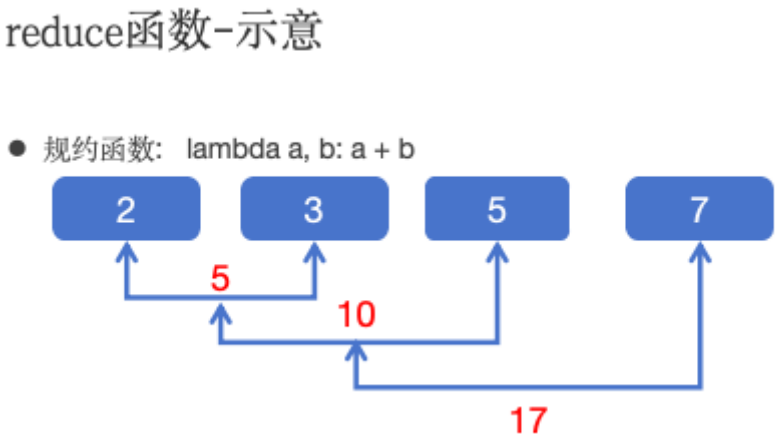
参数

- function：含有两个参数的函数。

- iterable：可跌代对象。
- initial：初始累积值（可选）。

reduce函数-示意

规约函数: `lambda a, b: a + b`



functools.reduce函数示例

```
from functools import reduce

data = [1, 2, 3, 4]

def add(x, y):
    return x + y

values = reduce(add, data)
print(values)
```

functools.partial函数

作用：

返回一个新的 部分对象，此对象可以固定部分参数，在调用部分对象时可以补齐参数再开始计算。

调用格式：

```
functools.partial(func, *args, **keywords)
```

参数

- func：实际调用函数
- `*args`, `**keywords`：其他可变参数

functools.partial函数示例

```
from functools import partial

# 创建线性函数 f(x) = ax + b
def linear(a, b, x):
    return a * x + b

# 固定a和b创建特定函数
line1 = partial(linear, 2, 1)  # f(x) = 2x + 1
print(line1(x=3))  # 输出 7
```

练习

求100以内所有素数的乘积数，即求：2 * 3 * 5 * 7 * ... * 97 = ?

参考答案

```
from functools import reduce
def is_prime(x):
    if x <= 1:
        return False
    for i in range(2, x):
        if x % i == 0:
            return False
    return True

print(reduce( lambda a, b: a*b, filter(is_prime, range(100))))
```

第二十五章、函数（高级）

1. globals和locals函数

局部变量和全局变量

局部变量

- 在函数内部定义的变量称为局部变量(函数的形参也是局部变量)。

全局变量

- 在函数外部、.py文件的内部定义的变量称为全局变量。

globals() / locals() 函数

函数	说明
globals()	返回实现当前模块命名空间的字典。
locals()	返回一个代表当前局部符号表的字典。

返回值：字典（以变量名称作为键，而以其当前绑定的引用作为值）。

示例

```
a = 1
b = 2
c = 3
def fn(c, d):
    e = 300
    print("locals() 返回:", locals())
    print("globals() 返回:", globals())

fn(100, 200)
print("globals() 返回:", globals())
```

2. 动态执行Python

2.1 eval 函数

什么是动态执行

动态执行（Dynamic Execution）是指在 运行时动态解析、编译并执行 Python 代码，而不是在编写代码时静态定义。

动态执行相关的内置函数：

1. eval() —— 执行 单个表达式 并返回结果。
2. exec() —— 执行 代码块（程序段），不返回结果。
3. compile() —— 将代码编译为字节码，供 eval() 或 exec() 执行。

动态执行相关函数

函数	说明
eval(source, globals=None, locals=None)	source参数将作为一个 Python 表达式，使用 globals 和 locals 映射作为全局和局部命名空间被解析并求值。
exec(source, globals=None, locals=None)	source参数将作为一个 Python 程序，使用 globals 和 locals 映射作为全局和局部命名空间被解析并求值。
compile(source, filename, mode)	将代码字符串编译为字节码，供 eval() 或 exec() 执行。

eval()函数

作用：

执行一个 字符串形式的 Python 表达式，并返回计算结果。

适用场景：

计算简单的表达式、函数调用等。

示例

```
# eval函数示例

x = 100
y = 200

expr = 'x + y'

# print(eval(expr))  # 300
# print(eval(expr, {'x':10, 'y':20}))  # 30
print(eval(expr, {'x':10, 'y':20}, {'x':1}))  # 21
```

2.2 exec 函数

exec()函数

作用：

执行 字符串形式的 Python 代码块（如 if、for、def），只返回None。

适用场景：

动态执行多行代码、定义函数、修改变量等。

函数

函数	说明
exec(source, globals=None, locals=None)	source参数将作为一个 Python 程序，使用 globals 和 locals 映射作为全局和局部命名空间被解析并求值。

示例

```
# exec函数示例

prog = '''
x = 100
y = 200
z = x + y
print('x:', x, 'y:', y, 'z:', z)
'''

# value = exec(prog)  # 当前模块内执行
# print('value:', None)  # None
# print(x, y, z)
# glb = {}
# exec(prog, glb)  # glb环境内执行
# print("全局字典:", glb)
loc = {}
```

```
glb = {}
exec(prog, glb, loc)  # 在全局为gbl, 局部为loc的函数内部执行
print("全局字典:", glb)
print("局部字典:", loc)
```

2.3 compile 函数

作用：

将代码字符串编译为字节码，供 eval() 或 exec() 执行。

适用场景：

提高重复执行的效率。节约编译时间。

函数

函数	说明
compile(source, filename, mode)	将代码字符串编译为字节码，供 eval() 或 exec() 执行。

示例

```
# compile函数示例

# 编译程序
prog = '''
x = 100
y = 200
z = x + y
print('x:', x, 'y:', y, 'z', z)
'''
prog_obj = compile(prog, '<string>', 'exec')
exec(prog_obj)
# 编译表达式
expr = 'x + y'
expr_obj = compile(expr, '<string>', 'eval')
value = eval(expr_obj)
print('value:', value)
```

Python动态执行的优缺点

优点

1. 可以根据当前状态，动态生成代码并运行，灵活性高。
2. 适合实现插件系统、脚本扩展等需求。

缺点

1. 安全风险。
2. 性能较低。
3. 调试困难。
4. 维护成本高。
5. 版本升级后兼容性测试困难。

3. 函数嵌套定义

什么是函数嵌套定义

函数嵌套定义是指一个函数里用def语句来创建其它函数的情况。

示例

```
# 此示例示意函数的嵌套定义

def fn_outer():
    print("fn_outer被调用")
    def fn_inner():
        print("fn_inner被调用")
    # 调用嵌套函数fn_inner
    fn_inner()
    return fn_inner

inner_fun = fn_outer()
print("fn_outer调用结束")
inner_fun()
```

4. python作用域

什么是作用域?

作用域指的是变量、函数和类等标识符在代码中的可访问区域。

Python主要有四种作用域:

1. 局部作用域（Local）；
2. 嵌套函数作用域（Enclosing Function Local）；
3. 全局作用域（Global）；
4. 内置作用域（Built-in）；

python作用域如图所示:

变量名的查找规则

1. 在访问变量时，先查找本地变量，然后是包裹此函数外部的函数内部的变量，之后是全局变量，最后是内置变量。
2. 在默认的情况下，变量名赋值会创建或者改变本地作用域变量。

python作用域示例

```
v = 100
def fun1():
    v = 200  # 创建fun1内部的局部变量v
    print("fun1.v =", v)
    def fun2():
        v = 300  # 创建fun2内部的局部变量v
        print("fun2.v =", v)
    fun2()

fun1()
print("v =", v)
```

5. nonlocal语句

python中与作用域相关的语句有两条：

1. global 语句;
2. nonlocal 语句;

nonlocal语句

作用：

告诉解释器，nonlocal声明的变量不是局部变量,也不是全局变量,而是外部嵌套函数内的变量。

语法:

```
nonlocal 变量名1, 变量名2, ...
```

语法说明

1. nonlocal 语句只能在被嵌套的函数内部进行使用;
2. 对nonlocal变量进行赋值将对外部嵌套函数作用域内的变量进行操作;
3. 当有两层或两层以上函数嵌套时，访问nonlocal变量只对最近一层的变量进行操作;
4. nonlocal语句的变量列表里的变量名，不能出现在此函数的形参列表中。

示例

```
v = 100
def f1():
    v = 200
    print("f1:", v)
    def f2():
        nonlocal v
        v = 201 # 将f1函数内的v修改为201
        print("f2:", v)
    f2()
    print("f1 after f2()", v)
f1()
```

6. 类型注解

什么是类型注解

类型注解是开发者为变量、类属性、函数的形参或返回值指定预期的类型。

- 类型注解是可选的而不是 Python 的强制要求，但它们对 静态类型检查器 很有用处。 它们还能协助 IDE 实现代码补全与重构。

示例

```
home: str = '地球'

def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

类型注解的种类:

1. 变量标注。

2. 属性标注。
3. 函数标注。

变量或类属性的标注

示例

```
# 变量标注和类属性标注示例

count: int = 0

class Dog:
    home: str = '地球'

count = 3.14

print(count)
print(Dog.home)
```

函数标注

针对函数形参或返回值的标注。

示例

```
# 函数标注示例
def sum_two_numbers(a: int, b: int) -> int:
    return a + b

print(sum_two_numbers(100, 200))

def add_two_element(x: int | str, y: int | str) -> int | str:
    return x + y

print(add_two_element(1, 2))
print(add_two_element('ABC', 'DEF'))

print(add_two_element(3.14, 0.618))
```

typing — 对类型提示的支持

模块：typing

作用

用于支持复杂类型的类型注解（Type Hints）

typing 中定义的常用类型

类型/语法	说明
int, str, bool	基础类型（直接使用 Python 内置类型）。
List[int]	列表，元素为 int 类型
Dict[str, int]	字典，键为 str，值为 int。
Tuple[int, str]	元组，固定长度和类型。
Union[int, str]	允许 int 或 str 类型（
Optional[str]	等价于 Union[str, None]，表示可能为 None。
Callable[[int], str]	函数类型，参数为 int，返回 str。
Any	任意类型（禁用类型检查）。

示例

```
# typing 模块中定义的类型 示例

from typing import List

def mysum(data: List[int]) -> int:
    return sum(data)


lst = [1, 2, 3, 4, 5]
result = mysum(lst)
print('result:', result)
```

7. 函数的属性

函数的特殊属性

属性	说明
`__name__`	函数的名称（字符串形式）。
`__doc__`	函数的文档字符串（docstring）
`__module__`	函数定义所在的模块名（如果是当前模块，返回`"__main__"`）
`__defaults__`	函数的默认参数值（元组形式）。
`__annotations__`	函数的类型标注
`__closure__`	闭包引用的外部变量（如果是闭包）。

示例

```
# 此示例示意函数的特殊属性

def my_function(a: int=666, b: int=999) -> int:
    '''此函数用于累加两个整数，返回加法的结果！'''
    pass

print(my_function.__name__)  # 函数名
print(my_function.__doc__)  # 文档字符串
print(my_function.__module__)  # 模块名(当前为主模块)
print(my_function.__defaults__)  # 缺省参数的值(元组)
print(my_function.__annotations__)  # 函数的类型标注
print(my_function.__closure__)  # 闭包引用的外部变量
```

第二十六章、闭包

1. 闭包（Closure）

什么是闭包

闭包是指一个函数（称为嵌套函数）引用了其外部函数作用域内的变量，并且该外部函数已经执行完毕（即其作用域理论上应该被销毁），但嵌套函数仍然可以访问这些变量。闭包的核心特点是“跨作用域引用”。

在本质上，闭包是将内部嵌套函数和函数外部的执行环境绑定在一起的对象。

闭包形成条件

闭包的三个必要条件：

1. 一个函数内部定义了另一个函数。
2. 嵌套函数内部引用了外部函数的变量。
3. 外部函数返回嵌套函数（或将其传递到其他作用域）。

闭包示例

```
def make_power(y):
    def fn(x):
        return x ** y
    return fn

pow2 = make_power(2)
print("5的平方是:", pow2(5))

pow3 = make_power(3)
print("6的立方是:", pow3(6))
```

闭包的优缺点

优点

1. 闭包可以替代全局变量，将数据隐藏在函数内部，避免污染全局命名空间。
2. 闭包可以记住函数调用之间的状态，适用于需要记忆功能的场景。
3. Python 的装饰器就是基于闭包实现的，可以动态增强函数功能。

缺点

1. 闭包会长期持有外部变量，可能导致内存无法释放。
2. 过度使用闭包可能导致代码难以理解，尤其是多层嵌套时。
3. 闭包的变量隐藏在嵌套函数中，调试时可能难以追踪。

第二十七章、装饰器

什么是装饰

装饰器是一个函数（或类），主要作用是来用包装另一个函数或类。

作用

在不修改被装饰的函数的源代码和调用方式的情况下，添加或改变原函数（或类）的功能。

装饰器的种类（两种）：

1. 函数装饰器
   - 函数装饰器是用来包装另一个函数或类的函数。
2. 类装饰器
   - 类装饰器是用来包装另一个函数或类的类。

1. 装饰器的原理和语法

装饰器语法

```
def 装饰器函数名(fn):
    语句块
    return 函数对象

@装饰器函数名1 <换行>
def 被装饰函数名(形参列表):
    语句块
```

示例

- 最初的函数

```
def do_somthing():
    print('上班!')

do_somthing()  # ???
```

请问打印结果是什么?

- 实现替换

```
def take_a_holiday():
    print('度假! ')

def replace_deco(fn):
    return take_a_holiday

def do_somthing():
    print('上班!')

do_somthing = replace_deco(do_somthing)

do_somthing()  # ???
```

请问上述程序的打印结果是什么? 为什么?

- 使用装饰器

```
def take_a_holiday():
    print('度假! ')

def replace_deco(fn):
    return take_a_holiday
@replace_deco
def do_somthing():
    print('上班!')

# do_somthing = mydeco(do_somthing)

do_somthing()  # ???
```

在试一下，打印结果是什么?

这就是装饰器的语法和原理。

2. 没有参数的函数的装饰器

本节目标:

1. 在装饰器内部调用原函数。
2. 能够为函数提供附加的功能。
3. 理解装饰器包装函数的原理。

示例

使用装饰器返回的函数内回调被装饰函数。

```
def replace_deco(fn):
    def take_a_holiday():
        print('购机票! ')
        fn()
        print('度假! ')
    return take_a_holiday

@replace_deco
def do_somthing():
    print('上班!')

do_somthing()  # ???
```

上述程序 形参fn绑定的是被装饰函数do_somthing。

do_somthing 重新绑定了 take_a_holiday 函数，在调用do_somthing时，会在调用fn的前后执行相应的打印，从而实现了对原函数加“壳”的操作。

3. 带有参数和返回值的函数的装饰器

当被装饰函数有参数和返回值，则装饰器返回的函数也必须有同样的参数和返回值。

本节目标:

- 理解带有参数和返回值的函数的装饰器定义方法。
- 理解装饰器内包装函数的参数和返回值的传递顺序。

```
arguments = []
result_numbers = []

def recorder_operation_deco(fn):
    '''此装饰器实现将被装饰函数的参数形成元组(n1, n2) 放入 arguments列表
    将运 行的结果放入 result_numbers 列表。'''
    def operations(n1, n2):
        arguments.append((n1, n2))
        r = fn(n1, n2)
        result_numbers.append(r)
        return r
    return operations

@recorder_operation_deco
def myadd(a, b):
    return a + b

@recorder_operation_deco
def mymul(x, y):
    return x * y
```

```
print('1 + 2 =', myadd(1, 2))
print('3 * 4 =', mymul(3, 4))

print('所有的参数:', arguments)  # [(1, 2), (3, 4)]
print('所有的结果:', result_numbers)  # [3, 12]
```

上述程序用，因为被装饰函数 myadd 和 mymul 都有两个形参。因此装饰器函数返回的函数 operations 也必须能接收两个实参。否则调用出错。

4. 带有不定长参数的函数的装饰器

知识点回顾

函数形参的定义方式：

- 星号元组形参（*args） 用来收集多余的位置传参。
- 双星号字典形参（**kwargs） 用来收集多余的关键字传参。

函数传参的方式：

- 位置传参
  - 拆解序列进行位置传参。

```
lst = [1, 2, 3]
print(*lst)  # 等同于 print(1,2,3)
```

- 关键字传参
  - 拆解字典进行关键字传参。

```
func(**{'a':1, 'b':2})
# 等同于 func(a=1, b=2)
```

以上是《python编程语言基础篇》的内容，这里不再赘述。

带有不定长参数的函数的装饰器

当被装饰函数有不定长参数和返回值时，则装饰器返回的函数的参数通常使用 `*args` 和 `**kwargs` 来接收所有的传参。

在调用被装饰函数时，也使用 `*args` 和 `**kwargs` 来拆解 args 元组和 kwargs 字典进行位置传参和关键字传参。

```
def mydeco(fn):
    '''定义一个 包裹被装饰函数的装饰器，此装饰器在调用函数之前或之后都打印函数名！'''
    def wrap(*args, **kwargs):
        print(f'调用：{fn.__name__} 之前')
        r = fn(*args, **kwargs)
        print(f'调用：{fn.__name__} 之后')
        return r
    return wrap

@mydeco
def myadd(a, b, c=0, d=0):
    return a + b + c + d

@mydeco
def mypower_mod(a, b, *, mod=0):
    if mod:
        return a ** b % mod
    return a ** b

print(myadd(1, 2, 3))  # 6
print(mypower_mod(5, 2))  # 25 # 计算 5的平方
print(mypower_mod(4, 3, mod=5))  #4 # 计算 4的立方再对5求余数
```

上述装饰器 mydeco 可以装饰任何的函数，并能够完美调用被装饰函数，且能在调用被装饰函数之前执行相应的自定义代码。

5. 带参数的装饰器

语法

```
@装饰器 <换行>
def 被装饰函数名(形参列表):
    语句块
```

说明

装饰器是个表达式，此表达式一定要返回带有一个参数的可被调用的函数（或对象）。

例如

```
def mydeco(fn):
    def wrap(*args, **kwargs):
        ...
    return wrap
```

示例

```
# 写一个装饰器，让被装饰函数重复调用指定的次数n

def repeat(n):
    def mydeco(fn):
        def wrap(*args, **kwargs):
            for x in range(n):
                fn(*args, **kwargs)
        return wrap
    return mydeco


@repeat(5)
def welcome(name):
    print('你好:', name)

@repeat(3)
def bye():
    print('再见')

welcome('Python!')
bye()
```

上述程序中 repeat(5) 实际是一个函数调用，此函数调用将返回一个装饰器。

6. 装饰器应用案例之权限管理

目标

使用装饰器实现权限管理规则，对银行存钱（savemoney）和取钱（withdraw）业务的实际操作函数的调用进行管理。

示例代码

```
# 装饰器应用案例之权限管理

bank_account = {
    '魏明择': {'password': '123456', 'money': 500},
    '小张': {'password': '654321', 'money': 10000},
}

def privileged_check(fn):
    '权限管理装饰器'
    def wrap(name, x):
        if name not in bank_account:
            raise ValueError(f'账户异常')
        password = input(f'请输入{name}的密码: ')
        if password != bank_account[name]['password']:
```

```
            raise ValueError(f'账户异常')
        return fn(name, x)
    return wrap

@privileged_check
def savemoney(name, x):  # 存钱
    bank_account[name]['money'] += x
    print(name, "存钱", x, "元。")

@privileged_check
def withdraw(name, x):  # 取钱
    bank_account[name]['money'] -= x
    print(name, "取钱", x, "元。")

print('原来的银行账户:', bank_account)
savemoney("魏明择", 200)
savemoney("小张", 20000)
withdraw("魏明择", 300)
print('现在的银行账户:', bank_account)
```

上述程序中，bank_account 是模拟银行账户信息，键是账户名称，'password' 是账户密码，'money' 的值是账户余额。

装饰器在为调用存钱 savemoney 和取钱 withdraw 函数时，添加了验证是否存在此账户和验证密码的操作，如果出错就会引发异常，从而不会对被包装函数进行调用，达到了不让非法用户进行操作的目的。

7. 装饰器的嵌套装饰

目标

使用装饰器，对银行存钱（savemoney）和取钱（withdraw）业务的实际操作函数成功调用后模拟发送短信息。

对一个函数可以添加多个装饰器。

装饰器的嵌套装饰的语法

```
@装饰器n <换行>
...
@装饰器2 <换行>
@装饰器1 <换行>
def 被装饰函数名(形参列表):
    语句块
```

说明

- 距离被装饰函数最近的装饰器被优先装饰，如上面语法所示：
- 装饰器1最先替换被装饰函数名绑定的函数。
- 装饰器2再替换被装饰函数名绑定的函数。以此类推。

上述语法等价于

```
@装饰器n <换行>
...
@装饰器2 <换行>
@装饰器1 <换行>
def 被装饰函数名(形参列表):
    语句块

# 等价于
被装饰函数名 = 装饰器2(装饰器1(被装饰函数名))
```

示例

```
# 此示例示意 装饰器的嵌套装饰 的用法

bank_account = {
    '魏明择': {'password': '123456', 'money': 500},
    '小张': {'password': '654321', 'money': 10000},
}

def message_send(fn):
    def wrap(name, x):
        r = fn(name, x)
        print(name, '您办理了银行的业务，短息发送中...')
        return r
    return wrap

def privileged_check(fn):
    '权限管理装饰器'
    def wrap(name, x):
        if name not in bank_account:
            raise ValueError(f'账户异常')
        password = input(f'请输入{name}的密码: ')
        if password != bank_account[name]['password']:
            raise ValueError(f'账户异常')
        return fn(name, x)
    return wrap


@message_send
@privileged_check
def savemoney(name, x):  # 存钱
```

```
    bank_account[name]['money'] += x
    print(name, "存钱", x, "元。")

@privileged_check
def withdraw(name, x):  # 取钱
    bank_account[name]['money'] -= x
    print(name, "取钱", x, "元。")

print('原来的银行账户:', bank_account)
savemoney("魏明择", 200)
savemoney("小张", 20000)
withdraw("魏明择", 300)
print('现在的银行账户:', bank_account)
```

上述程序实现了对savemoney 添加了 权限管理的装饰器 privileged_check 后，又添加了message_send装饰器。

8. 装饰类的装饰器

装饰器的实质

替换被装饰函数（或类）的变量，让他重绑定一个函数（或者类）。

用装饰器可以装饰类型:

1. 函数
2. 类
3. 方法

Python的类名也是变量，也可是使用装饰器来替换。

装饰类的装饰器语法

```
@装饰器n <换行>
...
@装饰器2 <换行>
@装饰器1 <换行>
class 类名:
    语句块
```

示例

```
# 装饰类的装饰器 示例
```

```
def replace_animal(fn):
    class Cat:
        def speak(self):
            print('喵')
    return Cat

@replace_animal
class Dog:
    def speak(self):
        print('旺')

dog = Dog()
dog.speak()
```

上述装饰器将 Dog 变量重新绑定成了 `class Cat`。

9. __call__方法

`__call__` 是一个特殊方法，它允许一个类的对象（也叫实例）像函数一样被调用。

当一个类实现了 `__call__` 方法后，该类的对象就成为了可调用对象，可以通过对象名() 的方式调用该方法，而实际上执行的是 `__call__` 方法中的代码。

示例

```
class NumberAdder:
    def __init__(self, start=0):
        self.value = start
    def __call__(self, a_number):
        self.value += a_number

adder = NumberAdder(0)
adder.__call__(100)
adder.__call__(200)
adder(300)
print(adder.value)
```

上述示例中，adder 是对象，但它可以像函数一样调用，并能够返回值（尽管此示例返回 None）。

10. 类装饰器

什么是类装饰器

类装饰器（Class Decorator） 是使用类（而不是函数）来装饰其他函数或类的装饰器。

类装饰器通过实现 `__call__` 方法，使得类的实例可以像函数一样被调用，从而实现对目标函数或类的增强或修改。

类装饰器的基本结构：

在 `__init__` 方法中接收被装饰的函数或类。 在 `__call__` 方法中定义装饰逻辑，并调用原函数或类。

示例

```
# 类装饰器示例

class NumberAdder:
    def __init__(self, fn):
        self.fn = fn
        self.value = 0

    def __call__(self, *args):
        result = self.fn(*args)
        self.value += result
        return self.value

@NumberAdder
def add(a, b, c=0, d=0):
    return a + b + c + d

# add = NumberAdder(add)

print(add(1, 2))  # 3

# 能否将之前加过的和也累加到此函数的调用?
print(add(3, 4, 5))
```

上述程序中 NumberAddr 就是一个类，同时它也可以作为一个装饰器使用。

函数装饰器 和 类装饰器 的比较

- 函数装饰器通常使用闭包来存储被装饰函数等信息，简单方便。
- 类装饰器通过属性可以存储更多的信息，方便实现更复杂的功能。

后面学到的很多装饰器都是类装饰器，如：

1. classmethod
2. staticmethod
3. property

第二十八章、类和对象（高级）

1. 对象属性管理的内置函数

示例：

```
class Dog:
    pass

dog1 = Dog()
dog1.color = "黑色"
```

问题：

一个小狗类型的对象 dog1 它是否有 kind 这个属性呢?

上述程序为dog1绑定的对象创建了color 属性，但是在运行程序后，无法动态的去检测 是否有 kind 属性。

如果需要动态检测、添加、删除和修改属性，可以使用对象属性管理的内置函数。

对象属性管理的内置函数如下表所示

函数	说明
getattr(obj, name[, default])	从一个对象得到对象的属性；getattr(obj, 'y') 等同于x.y; 当属性不存在时，如果给出default参数，则返回default,如果没有给出default 则产生一个AttributeError错误。
setattr(obj, name, value)	与getattr函数相对应，给对象obj的名为name的属性设置相应的值value, setattr(obj, 'y', v) 等同于 obj.y = v。
hasattr(obj, name)	实参是一个对象和一个字符串name。如果对象存在属性 name，则返回 True，否则返回 False。，此种做法可以避免在getattr(obj, name)时引发错误。
delattr(obj, name)	这是 setattr() 的相关函数。 实参是一个对象和一个字符串 name。 其中字符串name必须是对象的某个属性的名称。 该函数会删除指定的属性。 例如， delattr(obj, 'y') 等同于 del obj.y。

示例

```
class Dog:
    pass

dog1 = Dog()
dog1.color = "黑色"

v = getattr(dog1, 'color', 'xxxxx')  # 等同于 v = dog1.color
v = getattr(dog1, 'kinds')  # 出错，没有dog1.kinds属性
v = getattr(dog1, 'kinds', '没有这个属性')  # v= '没有这个属性'
print("v:", v)
print(hasattr(dog1, 'color'))  # True
print(hasattr(dog1, 'kinds'))  # False
setattr(dog1, 'kinds', '京巴')  # 等同于dog1.kinds ='京巴'
print(dog1.kinds)
print(hasattr(dog1, 'kinds'))  # True
delattr(dog1, 'kinds')  # 等同于 del dog1.kinds
print(hasattr(dog1, 'kinds'))  # False
```

2. 对象的特殊属性

`__class__` 属性

作用:

用于绑定创建此对象的类。

> 可以借助于此属性来访问创建此对象（实例）的类。

示例

```
class Dog:
    '''Dog是人类最亲近的小动物'''
    def __init__(self, c, k):
        self.color = c  # 颜色
        self.kind = k  # 品种

dog1 = Dog('黄色', '金毛')

# __class__ 属性
print(dog1.__class__)
print(dog1.__class__ is Dog)
```

运行结果

```
<class '__main__.Dog'>
True
```

`__dict__` 属性

作用：

用于绑定一个存储此对象自身属性和值的字典。

如果类内有 `__slots__` 列表，则对象不存在 `__dict__` 属性。

示例

```python
class Dog:
    '''Dog是人类最亲近的小动物'''
    def __init__(self, c, k):
        self.color = c  # 颜色
        self.kind = k  # 品种

dog1 = Dog('黄色', '金毛')

# __dict__ 属性
print(dog1.__dict__)

for attr in dog1.__dict__:
    print('属性:', attr, '值', getattr(dog1, attr))
```

运行结果

```
{'color': '黄色', 'kind': '金毛'}
属性: color 值 黄色
属性: kind 值 金毛
```

3. 类属性

Python 中一切皆对象，类也是一个对象，它相当于对象的工厂。

什么是类属性

类属性是类的属性，此属性属于类，不属于此类的对象（实例）。

作用：

通常用来存储该类创建的对象的共有属性。

语法

```
class 类名:
    类属性名 = 值  # 创建类时直接定义属性
    def 方法名(self, ...):
        ...
类名.类属性名 = 值  # 后添加类属性
```

示例：

```
class Dog:
    home = '地球'  # 创建类时直接创建属性
    def eat(self, food):
        pass

Dog.species = '动物'  # 后添加类属性

print(Dog.home)
print(Dog.eat)
print(Dog.species)

dog1 = Dog()
print(dog1.home)
print(dog1.__class__.home)

# dog1.home = '中国'
dog1.__class__.home = 'xxxxx'
print(Dog.home)
```

对于类属性，Dog 类和 Dog类的对象 dog1 都可以访问类的属性 home 。但对象 dog1 只有通过 `__class__` 属性才能修改类属性。

类属性说明

- 类属性，可以通过该类直接访问。
- 类属性，可以通过类的对象直接访问。
- 类属性可以通过此类的对象的 `__class__` 属性间接访问。
- 类的方法也是类属性。

4. 类的特殊属性

类也是一个对象，它有自己的特殊属性。

常用的特殊属性有:

`__doc__` 属性：用于绑定文档字符串。

`__base__` 属性，用于绑定此类的第一个基类。

`__dict__` 属性：用于绑定类属性的字典。

示例

```
class Animal:
    pass

class Dog(Animal):
    '''Dog是人类最亲近的小动物'''
    home = '地球'
    def eat(self, food):
        pass

# __doc__ 属性
print(Dog.__doc__)

# __base__ 属性
print(Dog.__base__)

# __dict__ 属性
print(Dog.__dict__)
for attr in Dog.__dict__:
    print('属性:', attr, '值', getattr(Dog, attr))
```

5. 类方法

类方法 @classmethod

什么是类方法

类方法是用于描述类的行为的方法，类方法属于类，不属于该类创建的对象。

说明

- 类方法需要使用 @classmethod 装饰器定义。
- 类方法至少有一个形参，第一个形参用于绑定类，约定写为 cls。
- 类和该类的对象都可以调用类方法。
- 类方法不能访问此类创建的对象的属性。

示例

```python
class Car:
    total_count = 0  # 此类属性用于记录所有车对象的数量。
    def __init__(self, brand, model):
        self.brand, self.model = brand, model
        self.__class__.total_count += 1  # 总数加1
        print(self.brand, self.model, '被创建')

    def __del__(self):
        self.__class__.total_count -= 1  # 总数减1
        print(self.brand, self.model, '被销毁')

    @classmethod
    def get_total_count(cls):
        return cls.total_count


car1 = Car('比亚迪', '秦')
car2 = Car('小米', 'SU7')
print(Car.get_total_count())
del car2
print(car1.get_total_count())
```

上述程序中， total_count 只有一个，他属于类Car，它的对像可以对其取值，但要对其赋值则必须通过 `__class__` 属性进行操作。

get_total_count 是类方法，他需要通过 类 Car 或 Car 类的对象调用，但在方法内只能通过 cls 属性访问类，但不能访问调用此方法的对象。

6. 静态方法

静态方法 @staticmethod

什么是静态方法

静态方法是定义在类的内部函数，此函数的作用域是类的内部。

说明

- 静态方法需要使用 @staticmethod 装饰器定义。
- 静态方法与普通函数定义相同，不需要传入 self 对象参数和 cls 类参数。
- 静态方法只能凭借该类或类创建的对象调用。
- 静态方法不能访问类属性和对象属性。

示例

```
class Mathematics:
    '''数学相关的类'''

    @staticmethod
    def myadd(x, y):
        return x + y

m = Mathematics()  # 创建一个数学类的对象

result = m.myadd(100, 200)
print(result)
result = Mathematics.myadd(1, 2)
print(result)
```

myadd(x, y) 是类 Mathematics 内的静态方法，类 Mathematics 和该类的对象都可以调用这个方法，但在方法内无法得知调用此方法的类和对象。

函数和方法总结

对象方法、类方法、静态方法、函数总结：

- 不访问类属性和实例属性，用静态方法。
- 只访问类属性，不访问对象属性用类方法。
- 即访问类属性，也访问对象属性用对象方法（也称作实例方法）。
- 函数与静态方法相同，只是静态方法的作用域定义在类内，需要用 `类名.函数名()` 或 `对象.函数名()` 的方式来调用。

第二十九章、面向对象（高级）

知识点回顾

什么是封装

封装是指将数据（属性）和操作数据的行为（方法）封装在类中，并控制外部对内部数据的访问权限，让用户通过特定的方法才能操作这些对象。

封装的目的

- 隐藏类的内部实现细节，让使用者不用关心这些细节，从而保证数据的安全性。
- 防止外部直接修改对象的内部状态。
- 提供清晰的接口（尽可能少的方法(或属性)）供外部使用。

封装的方法

- 无封装，公用属性，任何语句都可以直接属性和方法，如：

```
class Human:
     pass

h1 = Human()
h1.age = 18
```

- 私有属性和方法封装（以双下划线 __ 开头且不以双下划线 __ 结尾的属性和方法），如：

```
class Human:
    def __init__(self, n, a):
        self.__name, self.__age = n, a

h1 = Human()
h1.__age = 18  # 报错
```

1. __slots__列表

作用：

限定每一个对象只能有固定的属性，防止动态添加属性。

优化内存，提高访问速度。

- 使用列表来保存属性名和属性值，不适用字典 `__dict__` 保存，节省内存，提高速度。

说明：

`__slots__` 属性是类属性，他是一个列表，列表的值是字符串（属性名称）。

含有 `__slots__` 属性的类所创建的实例对象没有 `__dict__` 属性，即此实例不用字典来存储对象的属性（实例属性）。

示例

```
# __slots__ 列表 示例

class Human:
    __slots__ = ['name', 'age']
    def __init__(self, n, a):
        self.name, self.age = n, a

s1 = Human('小张', 18)
s1.score = 100  # 添加额外的属性报错
print(s1.__dict__)  # s1也没有 __dict__属性
```

2. 特性属性property

特性属性是封装方法之一。

什么是特性属性

特性属性（property）是模拟出来的类属性，它将对象对特性属性的操作映射成对方法调用。从而实现对属性操作的控制。

内置类 property 的构造函数

```
property(fget=None, fset=None, fdel=None, doc=None)
```

参数说明

- fget 绑定取值的方法，格式为：方法名(self):
- fset 绑定赋值的方法，格式为：方法名(self, new_value):
- fdel 绑定删除的方法，格式为：方法名(self):

- doc 绑定特性属性的文档字符串。

示例

第一步：写一个圆形的类。

```
class Circle:  # 圆类
    def __init__(self, r=1):
        self.radius = r  # 半径

c = Circle(10)
print('圆的半径: ', c.radius)
c.radius = -100
print('圆的半径: ', c.radius)
del c.radius
print('圆的半径: ', c.radius)
```

上述程序对 圆的半径 radius 赋值为 -100 是错误的。且用 del 语句可以删除 radius 属性，导入后面程序无法正常运行。

第二步：改进上述程序，将半径改为私用属性。然后用方法 set_radius 和 get_radius 对半径进行操作。

```
# 特性属性示例

class Circle:  # 圆类
    def __init__(self, r=1):
        self.__radius = r  # 半径
    def get_radius(self):
        return self.__radius
    def set_radius(self, r):
        if r < 0:
            raise ValueError('圆的半径不允许小于零! ')
        self.__radius = r
    def del_radius(self):
        print('不允许删除圆的半径')
    radius = property(get_radius, set_radius, del_radius, '半径')

c = Circle(10)
print('圆的半径: ', c.get_radius())
c.set_radius(-100)  # 报错
print('圆的半径: ', c.radius)
```

上述程序能保证半径属性不被删除，且不能把半径设置为负数。但方法调用比较麻烦，不如直接使用属性方便。

第三步：改进上述程序，创建一个特性属性 radius, 让其对此属性取值调用 get_radius，赋值调用 set_radius,删除调用 del_radius 什么都不做。

```
# 特性属性示例

class Circle:  # 圆类
    def __init__(self, r=1):
        self.__radius = r  # 半径
    def get_radius(self):
        return self.__radius
    def set_radius(self, r):
        if r < 0:
            raise ValueError('圆的半径不允许小于零! ')
        self.__radius = r
    def del_radius(self):
        print('不允许删除圆的半径')
    radius = property(get_radius, set_radius, del_radius, '半径')

c = Circle(10)
# print('圆的半径: ', c.get_radius())
# c.set_radius(-100)
print('圆的半径: ', c.radius)
# c.radius = -100
# print('圆的半径: ', c.radius)
del c.radius
print('圆的半径: ', c.radius)
```

使用 radius 属性既方便，有可以控制赋值的范围，完美解决上述所有的问题。

3. 特性属性装饰器

property 是内置类，它是一个类装饰器。

通过 property 装饰器，可以将被装饰方法名重新绑定为特性属性的操作。

装饰器

@property(fget=None) 或 @property.getter(fget=None)

- 装饰取值方法

@property.setter(fset=None)

- 装饰赋值方法

@property.deleter(fdel=None)

- 装饰删除方法

示例

```
# 特性属性装饰器的应用 示例
import math

class Circle:  # 圆类
    def __init__(self, r=1):
        self.__radius = r  # 半径

    @property
    def radius(self):
        return self.__radius

    @radius.setter
    def radius(self, r):
        if r < 0:
            raise ValueError('圆的半径不允许小于零! ')
        self.__radius = r
    @radius.deleter
    def radius(self):
        print('不允许删除圆的半径')

    @property
    def area(self):
        return math.pi * self.__radius ** 2

    @area.setter
    def area(self, area):
        self.__radius = math.sqrt(area / math.pi)

c = Circle(10)
# print('圆的半径: ', c.get_radius())
# c.set_radius(-100)
print('圆的半径: ', c.radius)
# c.radius = -100
# print('圆的半径: ', c.radius)
# del c.radius
# print('圆的半径: ', c.radius)

print('圆的面积:', c.area)
c.area = 200
print('圆的半径:', c.radius)
```

注意：第一次

```
@property
def radius(self):
    return self.__radius
```

后 radius 绑定的是一个 property 对象。之后的 radius.setter 都是 property对象的方法。

上述程序完美实现了上一节课一样的功能，且少了几个get_radius、set_radius 和 del_radius 属性来绑定方法。

练习

写一个正方形类，有三个属性:

1. 边长 length of a side
2. 周长 perimeter
3. 面积 area

用此类创建的对象，此三个属性其中一个属性改变，其他所有属性同步变化。

注：用特殊属性@property实现。

4. 描述符协议

什么是描述符协议

描述符协议(Descriptor Protocol)是Python中一种底层机制，它允许对对象的属性的取值、设置值和删除属性操作行为进行控制。从而实现更深层次的封装。

实现了描述符协议的类就是描述符。

描述符协议包含 `__get__` 、`__set__` 和 `__delete__` 三个核心方法，一个类只要实现了其中任意一个方法，就被视为描述符。

Python 中的 property、classmethod、staticmethod 都是描述符。

描述符协议的核心方法

描述符协议包含三个核心方法：

1. `def __get__(self, instance, owner):`
   - 用于控制属性的访问(获取)操作。
2. `def __set__(self, instance, value):`
   - 用于控制属性的赋值操作。
3. `def __delete__(self, instance):`
   - 用于控制属性的删除操作。

参数:

- instance是拥有该描述符的对象(如果是类访问则为None)。
- owner是拥有该描述符的类。
- value是要设置的值。

示例

第一步: 定义一个 AreaDescriptor 类，实现了描述符协议的三个方法，但是里面只有一句打印 print。

```
# 描述符协议 示例:
# https://weimingze.com

import math

class AreaDescriptor:
    def __get__(self, instance, owner):
        print('__get__(', instance, owner, ')')

    def __set__(self, instance, value):
        print('__set__(', instance, value, ')')

    def __delete__(self, instance):
        print('__delete__(', instance, ')')

class Circle:  # 圆类
    def __init__(self, r=1):
        self.radius = r  # 半径

    area = AreaDescriptor()

c = Circle(10)
a = c.area
print('a:', a)
c.area = 200
print(c.radius)
del c.area
```

上述程序对 c.area 取值则调用 `AreaDescriptor.__get__` 方法，赋值到用 `AreaDescriptor.__set__` 方法，删除调用 `AreaDescriptor.__delete__` 方法。

改进上述程序，定义 AreaDescriptor 的 初始化方法 `def __init__(self, fget=None, fset=None, fdel=None, doc=None):`，使用像 property 特性属性的构造函数一样，传入相应的取值、设置值和删除的函数。并用内部属性绑定。

```
# 描述符协议 示例:
# https://weimingze.com

import math

class AreaDescriptor:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.__getter_cb = fget
        self.__setter_cb = fset
        self.__delete_cb = fdel
```

```
    def __get__(self, instance, owner):
        print('__get__(', instance, owner, ')')
        if self.__getter_cb is None:
            raise AttributeError('此属性不可以取值!')
        return self.__getter_cb(instance)

    def __set__(self, instance, value):
        print('__set__(', instance, value, ')')
        if self.__setter_cb is None:
            raise AttributeError('此属性不允许设置值! ')
        self.__setter_cb(instance, value)

    def __delete__(self, instance):
        print('__delete__(', instance, ')')

class Circle:  # 圆类
    def __init__(self, r=1):
        self.radius = r  # 半径

    def get_area(self):
        return math.pi * self.radius ** 2

    def set_area(self, area):
        self.radius = math.sqrt(area / math.pi)

    area = AreaDescriptor(get_area, set_area)

c = Circle(10)
a = c.area
print('a:', a)
c.area = 200
print(c.radius)
del c.area
```

上述程序完善了 AreaDescriptor 的 `__get__` 并调用 Circel 类中的 get_area 实现了取值。同样完善了 AreaDescriptor的 `__set__`方法 并调用 Circel 类中的 set_area 实现了赋值操作。

这个 AreaDescriptor 就是一个描述符，他实现了对Circle 对象的属性 area 的取值和赋值进行了控制。

5. 多继承

单继承

单继承是指一个子类继承自一个基类。

多继承

多继承是指一个子类继承自两个或两个以上的基类。

继承的语法回顾

```
class 类名(基类1,基类2,...):
    语句块
```

说明:

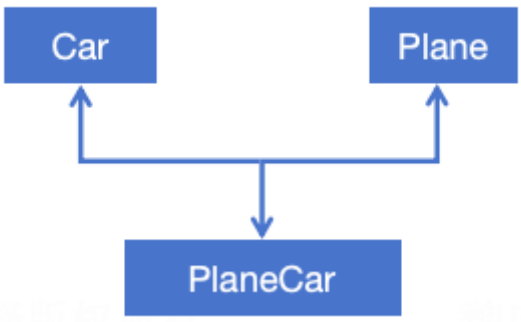
- 如果不写继承列表，则新类继承自object类
- 一个子类同时继承自多个父类，父类中的方法可以同时被继承下来。

示例

```
class Car:
    def run(self, speed):
        print("汽车以", speed, "km/h的速度行驶")

class Plane:
    def fly(self, height):
        print("飞行以海拔", height, "米的高度飞行")

class PlaneCar(Car, Plane):
    """PlaneCar类 ，同时继承自汽车和飞机"""

pc = PlaneCar()
pc.fly(10000)
pc.run(300)
```

上述程序中 pc 绑定了PlaneCar 类型的对象，且此对象已经拥有了两个方法 run 和 fly。

继承关系如图:



6. 方法解析顺序MRO

钻石继承（菱形继承）问题

示例

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass
```

上述程序的继承关系图：

```
  object
    |
    A
   / \
  B   C
   \ /
    D
```

上述程序构成了钻石继承的基本要素。下面我们为ABCD四个类添加方法 m 如下：

```
# 方法解析顺序MRO 示例
# https://weimingze.com

class A:
    'A类'
    def m(self):
        print('A')

class B(A):
    'B类'
    def m(self):
        print('B')

class C(A):
    'C类'
    def m(self):
        print('C')

class D(B, C):
    'D类'
    def m(self):
        print('D')

d = D()
d.m()  # 调用哪个方法?
```

上述程序打印结果为:D

多继承说明

- 一个子类同时继承自多个父类，父类中的方法可以同时被继承下来。
- 如果两个父类中有同名的方法，而在子类中又没有覆盖此方法时，调用结果难以确定。

去掉类D的方法m，如下

```
# 方法解析顺序MRO 示例
# https://weimingze.com

class A:
    'A类'
    def m(self):
        print('A')

class B(A):
    'B类'
    def m(self):
        print('B')

class C(A):
    'C类'
    def m(self):
        print('C')

class D(B, C):
    'D类'

d = D()
d.m()  # 调用哪个方法?
```

上述程序打印结果为:B，此时B的基类是A,那去掉 B 的 m方法试试。

去掉类B的方法m，如下.

```
# 方法解析顺序MRO 示例
# https://weimingze.com

class A:
    'A类'
    def m(self):
        print('A')

class B(A):
    'B类'

class C(A):
    'C类'
    def m(self):
```

```
        print('C')

class D(B, C):
    'D类'

d = D()
d.m()  # 调用哪个方法?
```

上述程序打印结果为:C，并不是调用B的基类A的 m方法。

上述程序调用m方法遵循的是方法解析顺序MRO。

什么是方法解析顺序

方法解析顺序(Method Resolution Order, MRO)是在多继承场景下确定方法调用顺序的规则。

作用:

当类继承自多个父类，且这些父类中有同名方法时，MRO决定了Python解释器查找方法的顺序。

__mro__属性

mro属性绑定一个元组，此元组记录的是此子类的所有基类。基类的先后顺序表示方法的查找次序。

super() 函数就是依据此属性对父类的方法进行查找的。

示例:

```
# 方法解析顺序MRO 示例
# https://weimingze.com

class A:
    'A类'
    def m(self):
        print('A')

class B(A):
    'B类'
    def m(self):
        print('B')

class C(A):
    'C类'
    def m(self):
        print('C')

class D(B, C):
    'D类'
    def m(self):
```

```
        print('D')

d = D()
d.m()  # 调用哪个方法?
print(D.__mro__)
```

打印结果是

```
D
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>)
```

这个元组就是MRO列表，他的先后顺序也是D类的方法查找顺序。

super() 函数在调用父类方法时，它也是依赖MRO列表来进行查找的。

7. object类

什么是object类

Python3 中的object 类是所有类的基类（根类），它是 Python 类继承体系中的最顶层类。所有类（无论是内置类还是用户自定义类）都直接或间接继承自 object。

在用class 语句创建类不写继承列表，则 Python 会默认让此类继承自 object。

object类的方法

object类提供一些共有的方法，供所有的子类使用。子类可以通过重写这些方法来覆盖 object 类的这些方法。

方法

1. `__init__(self)`：初始化方法。
2. `__new__(cls)`：创建对象的类方法。
3. `__str__(self)`：返回对象的字符串。
4. `__repr__(self)`：返回代表此对象的表达式字符串。

示例

```
>>> object
<class 'object'>

>>> class Dog:  # 自定义一个类
...     pass
```

```
...
>>> Dog.__base__  # 查看 Dog 类的父类为 object
<class 'object'>
>>> Dog.__base__.__base__  # 查看 Dog 类的父类的父类为None
>>> dir(object)  # 查看object 的所有属性，一些属性绑定的是方法。
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge
__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__in
it_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce
_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>>
```

8. __new__方法

什么是__new__方法

`__new__` 方法是一个负责创建对象的类方法（通常由解释器自动调用），而 `__init__` 则负责初始化对象。

作用：

控制对象的创建过程。

说明

此方法必须返回一个对象（实例）（通常是当前类的对象，也可以是其他类的对象）。

> 如果 `__new__` 方法未正确返回实例，则 `__init__` 不会被调用。

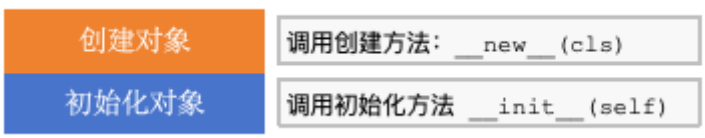
对象产生的顺序：

1. `__new__` 方法构造函数调用时最先被调用，它创建并返回一个对象。
2. 然后调用 `__init__` 对象方法对该对象进行初始化。

对象的创建

当调用构造函数：类名() 时，Python为创建对象做两件事:

1. 创建对象：调用 `__new__(cls)` 类方法来创建对象并返回此对象。
2. 初始化对象：调用 `__init__(self)` 方法类初始化对象self。



示例

```
# __new__ 方法 示例

class Pet:
    '''宠物类'''
    def __new__(cls, *args, **kwargs):
        print('__new__方法被调用, args:', args, 'kwargs:', kwargs)
        return super().__new__(cls)

    def __init__(self):
        print('__init__方法被调用')

ani = Pet()
print(ani)
```

运行结果

```
__new__方法被调用, args: () kwargs: {}
__init__方法被调用
<__main__.Pet object at 0x10f4e6e40>
```

从运行结果中，我们可以看出`__new__`方法会最先调用，然后调用 `__init__` 方法。

上述程序中 `__new__` 方法是调用 父类的 `__new__` 方法来创建对象，并将父类创建的对象返回。

再看下面的例子：

```
# __new__ 方法 示例

class Dog:
    def speak(self):
        print('旺')

class Cat:
    def speak(self):
        print('喵')

class Pet:
    '''宠物类'''
    def __new__(cls, *args, **kwargs):
        print('__new__方法被调用, args:', args, 'kwargs:', kwargs)
        return Dog()

    def __init__(self):
        print('__init__方法被调用')

ani = Pet()
print(ani)
ani.speak()
```

运行结果

```
__new__方法被调用, args: () kwargs: {}
__init__方法被调用
<__main__.Pet object at 0x10f4e6800>
旺
```

我们在 `__new__` 方法中返回一个 Dog类型的对象。那么创建的 Pet 类型的对象就被替换成了 Dog 类型的对象了。

`__new__` 和 `__init__` 方法的区别

1. `__new__` 方法是创建对象，`__init__` 是在创建完后初始化对象。
2. `__new__` 是类方法，传入的是类cls，`__init__` 是对象（实例）方法，传入的是对象self, 是 `__new__` 调用后返回来的对象。
3. `__new__` 方法必须返回对象，而 `__init__` 必须返回 None。

9. 单例模式

什么是单例模式

单例模式（Singleton Pattern）是在整个应用程序内部，在同一时刻只允许有一个实例对象存在的模式。

单例模式其核心思想是确保一个类只有一个实例对象，并提供一个全局访问点来访问该对象。

例如:

现实世界中，地球就是一个单例；你在某一家银行的账户也是一个单例。

> 这种模式常用于需要严格控制资源或共享状态的场景，例如数据库连接池、线程池、日志记录器等。

Python语言实现单例模式的方法

示例

先看一下不是单例的情况:

```
# 单例模式 示例
```

```
class Singleton:
    """单例的实现方法!"""
    def __init__(self, *args):
        print("开始对单例对象进行初始化")

obj1 = Singleton()
print(obj1)
obj2 = Singleton()
print(obj2)
print(obj1 is obj2)
```

运行结果

```
开始对单例对象进行初始化
<__main__.Singleton object at 0x10ee96e40>
开始对单例对象进行初始化
<__main__.Singleton object at 0x10efd0cd0>
False
```

两个对象 obj1 和 obj2 的 id 不同，不是单例

使用 重写 `__new__` 方法，实现单例模式

```
# 单例模式 示例

class Singleton:
    """单例的实现方法!"""
    __instance = None  # 用来绑定唯一的对象。

    def __new__(cls, *args, **kwargs):
        "用来创建一个对象;如果已经创建过了,那么将不再创建此类的对象,直接返回已前创建过的对
象。"
        if cls.__instance is None:
            print('开始创建单例对象!')
            cls.__instance = object.__new__(cls, *args, **kwargs)  # 没有创建过此
对象，则调用父类的__new__创建
        return cls.__instance

    def __init__(self, *args):
        print("开始对单例对象进行初始化")

obj1 = Singleton()
print(obj1)
obj2 = Singleton()
print(obj2)
print(obj1 is obj2)
```

运行结果

```
开始创建单例对象!
开始对单例对象进行初始化
<__main__.Singleton object at 0x10d406f90>
开始对单例对象进行初始化
<__main__.Singleton object at 0x10d406f90>
True
```

两个对象 obj1 和 obj2 的 id 相同，都是同一个对象。是单例模式。

10. 面向对象总结

面向对象的特征

- 封装：

防止外部代码修改对象的内部状态，实现数据安全。

- 继承：

实现代码复用，子类个性化的添加和修改父类的方法。

- 多态：

当调用同一个方法时，对不同的对象呈现不同的行为。

- 抽象：

让设计者只考虑类的定义和调用；具体实现要靠子类。

封装方法总结

Python的封装方法有如下几种：

1. 无封装，公用属性，任何语句都可以直接属性和方法。
2. 私有属性和方法封装（以 __ 开头且不以 __ 结尾的属性和方法）。
3. __slots__ 列表封装（限制属性个数）。
4. 特性属性封装（用对应方法接管对属性的赋值、取值和删除操作）。
5. 描述符协议封装（控制属性的赋值和取值等操作）。

继承总结

1. 单继承：一个父类，安全稳定。
2. 多继承：两个或两个以上的父类，同名的方法会冲突。需要了解MRO才能够掌控调用顺序。
    - 尽量使用super() 函数来调用父类的方法。

多态：略！

抽象：略！

第三十章、内置函数重载

1. str函数重载

什么是内置函数重载

内置函数重载是指通过实现特殊方法（如 `__len__`、`__str__`、`__int__` 等），让自定义类创建的对象支持len(obj)、str(obj)、int(obj)等 Python 内置函数的操作。

作用

让开发者自定义类和 Python 的内置类相兼容，可以使用内置函数len(x)、str(x)等对自定义类创建的对象统一操作。

str(obj) 函数的重载

内置函数 str

内置函数	对应的特殊方法	说明
`str(obj)`	`__str__()`	返回对象的字符串表示

示例

```
# str(x) 函数重载 示例

class Human:
    def __init__(self, name, age):
        self.name, self.age = name, age

    def __str__(self):
        return f'{self.age}岁的{self.name}'

h1 = Human('张三', 18)
s = str(h1)  # 实际调用 h1.__str__() 方法
print(s)

print(h1)  # 也是调用的 str(h1)
```

2. repr函数重载

repr(obj)函数

repr(obj) 返回一个附合 Python 语法规则且能代表此对象的表达式字符串。

通常:

```
eval(repr(obj)) == obj  # 结果为True。
```

自定义的类的repr(obj)函数通常返回该类的构造函数表达式的字符串或字面值。

repr(obj) 函数的重载

内置函数	对应的特殊方法	说明
`repr(obj)`	`__repr__()`	返回对象的字面值的字符串表示（通常可用于 `eval()` 重建）

示例

```
# repr(obj) 函数重载 示例

class Human:
    def __init__(self, name, age):
        self.name, self.age = name, age

    def __str__(self):
        return f'{self.age}岁的{self.name}'

    def __repr__(self):
        return 'Human('+repr(self.name)+ ',' + repr(self.age) + ')'


h1 = Human('张三', 18)
print(h1)
s1 = repr(h1)  # s1 = "Human('张三',18)"
print(s1)

h2 = eval(s1)
print(h2)
```

repr(obj)函数调用方法说明

1. repr(obj) 函数先查找 `obj.__repr__()` 方法，调用此方法并返回结果。
2. 如果 `obj.__repr__()` 方法不存在，则调用 object 类的对象的 `__repr__()` 方法显示 `<__main__.xxxClass object at 0x102a8ea20>` 格式的字符串。

str(obj)函数调用方法说明

1. str(obj)函数先查找 `obj.__str__()` 方法，调用此方法并返回结果。
2. 如果 `obj.__str__()` 方法不存在，则调用 `obj.__repr__()`方法并返回结果。
3. 如果 `obj.__repr__()` 方法不存在，则调用 object 类的对象的 `__repr__()` 方法显示 `<__main__.xxxClass object at 0x102a8ea20>` 格式的字符串。

printf风格的字符串格式化中的转换符

转换符	含意
'r'	使用 repr() 转换任何 Python 对象。
's'	使用 str() 转换任何 Python 对象。

示例

```
# printf风格的字符串格式化的转换符

print('%r' % h1)  # 打印: Human('张三',18)
print('%s' % h1)  # 打印: 18岁的张三
```

3. len和abs等函数重载

Python中可以重载的内置函数

内置函数	对应的特殊方法	说明
str(obj)	`__str__()`	返回对象的字符串表示
repr(obj)	`__repr__()`	返回对象的字面值的字符串表示（通常可用于 eval() 重建）
len(obj)	`__len__()`	返回对象的长度（如容器的元素数量）
abs(obj)	`__abs__()`	返回对象的绝对值
reversed(obj)	`__reversed__()`	返回反向迭代器
round(obj, n)	`__round__(n)`	返回 obj 舍入到小数点后 n 位精度的值
iter(obj)	`__iter__()`	返回一个迭代器，使对象可迭代
next(obj)	`__next__()`	获取迭代器的下一个值
int(obj)	`__int__()`	将对象转换为整数
float(obj)	`__float__()`	将对象转换为浮点数
complex(obj)	`__complex__()`	将对象转换为复数
bool(obj)	`__bool__()`	返回对象的布尔值（True 或 False）
hash(obj)	`__hash__()`	返回对象的哈希值（用于字典键、集合成员等）

内置函数重载说明

内置函数重载的方法的参数和返回值要求必须和内置函数原有的参数和返回值一致，否则可能出现不可预知的错误。

如:

len(obj) 函数

- 此函数必须返回整数。

示例

自定义一个一维向量类。存入一些列整数。使用 len(obj)等函数来取值。这里len(obj) 函数必须返回整数，其他函数没有限制。

```
# len(obj) 和 abs(obj) 函数的重载 示例
class MyVector:
```

```
    '''自定义一维向量类。'''
    def __init__(self, values):
        self.data = list(values)

    def __len__(self):
        return len(self.data)

    def __abs__(self):
        return sum(self.data)

    def __round__(self, n=None):
        return 3.14

    def __reversed__(self):
        return self.data[::-1]

v = MyVector((5, -2, -4, 3))

print(len(v))
print(abs(v))
print(reversed(v))
print(round(v, 2))
```

4. 数值转换函数重载

内置函数	对应的特殊方法	说明
int(obj)	`__int__()`	将对象转换为整数
float(obj)	`__float__()`	将对象转换为浮点数
complex(obj)	`__complex__()`	将对象转换为复数
bool(obj)	`__bool__()`	返回对象的布尔值（True 或 False）

数值转换函数重载说明

数值转换函数重载的方法的返回值类型必须和函数名类型一致。

示例

```
# 数值转换函数重载 示例

class ElectricCar:
    def __init__(self, brand, eq):
        self.brand = brand  # 品牌
        self.e_quantity = eq  # 电量

    def __int__(self):
        return int(self.e_quantity)
```

```
    def __float__(self):
        return float(self.e_quantity)

    def __complex__(self):
        return float(self.e_quantity) + 2j

    def __bool__(self):
        return self.e_quantity > 5

ec = ElectricCar('比亚迪', 2.1234)

print('int(ec):', int(ec))
print('float(ec):', float(ec))
print('complex(ec):', complex(ec))
print('bool(ec):', bool(ec))

if ec:
    print('ec为真! ')
else:
    print('ec为假! ')
```

complex 函数重载说明

1. complex(obj) 方法优先取 `__complex__(self)` 方法的返回值作为结果返回。
2. 如果自定义对象内没有 `__complex__(self)` 方法，则会用 `__float__(self)` 方法的返回值作为实部，用0j作为虚部返回。
3. 如果再没有 `__float__(self)` 方法则会触发 TypeError 类型的错误并进入异常状态。

Python中任何对象都一定能取布尔值。

布尔测试函数重写说明：

1. 当自定义类内有 `__bool__(self)` 方法时，以此方法的返回值作为 bool(obj) 的返回值。
2. 当不存在 `__bool__(self)` 方法时，bool(x)返回 `__len__(self)` 方法的返回值是否为零来测试布尔值。
3. 当不存在 `__len__(self)` 方法时，则直接返回True。

5. callable函数

作用

callable() 函数用于动态检查一个对象是否可以被调用（即是否实现了 `__call__` 方法）。如果对象是可调用的（如函数、方法、类或实现了 `__call__` 的对象），则返回 True，否则返回 False。

调用格式

内置函数	对应的特殊方法	说明
callable(obj)	`__call__()`	检测一个是否能像函数一样通过括号() 调用。

示例

Dog 类内没有 `__call__` 方法时，Dog() 创建的对象不可以调用。

```
# callable函数 示例

class Dog:
    pass

dog1 = Dog()

print(callable(len))  # True
print(callable(Dog))  # True
print(callable(dog1))  # False

if callable(dog1):
    dog1()
    dog1.__call__()
```

运行结果

```
True
True
False
```

Dog 类内有 `__call__` 方法时，Dog() 创建的对象可以调用。

```
# callable函数 示例

class Dog:
    pass
    def __call__(self, *args, **kwargs):
        pass

dog1 = Dog()

print(callable(len))  # True
print(callable(Dog))  # True
print(callable(dog1))  # True

if callable(dog1):
```

```
    dog1()
    dog1.__call__()
```

运行结果

```
True
True
True
```

第三十一章、运算符重载

1. 算术运算符重载

什么是运算符重载?

运算符重载是指通过自定义类中的特殊方法来让自定义的类创建的对象可以使用运算符（如 +、- 等）进行操作。

作用

- 让类的便于使用。
- 让代码简洁易读。
- 重新定义运算符的运算规则。

算术运算符重载

方法名和参数	运算符和表达式	说明
`__add__(self, other)`	self + other	加法
`__sub__(self, other)`	self - other	减法
`__mul__(self, other)`	self * other	乘法
`__truediv__(self, other)`	self / other	除法
`__floordiv__(self, other)`	self // other	整数（地板除）
`__mod__(self, other)`	self % other	求余(取模)
`__pow__(self, other)`	self ** other	幂
`__matmul__(self, other)`	self @ other	矩阵乘法

说明:

- 运算符重载方法的参数已经有固定的含义，不建议改变原有的意义
- 重载后的运算符，不会改变运算符的优先级。

二元运算符重载方法格式:

```
def __xxx__(self, other):
    ....
```

> 对于算术运算符， self参数绑定运算符左侧的对象，other 绑定运算符右侧的对象。

示例

```
# 算术运算符重载 示例
# http://weimingze.com

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __add__(self, other):
        return Vector3D(self.x + other.x,
                        self.y + other.y,
                        self.z + other.z)

v1 = Vector3D(1, 2, 3)
v2 = Vector3D(4, 5, 6)
print("v1:", v1)
print("v2:", v2)
v3 = v1 + v2  # 等同于 v3 = v1.__add__(v2)
print('v3:', v3)
```

运行结果

```
v1: Vector3D(1,2,3)
v2: Vector3D(4,5,6)
v3: Vector3D(5,7,9)
```

2. 反向算术运算符重载

当运算符的左侧为内置类型，右侧为自定义类型进行算术运算符运算时，会出现TypeError错误。因无法修改内置类型的代码来实现运算符重载，此时需要使用反向算术运算符的重载来完成重载。

反向算术运算符重载的方法

方法名	运算符和表达式	说明
`__radd__(self, other)`	other + self	加法
`__rsub__(self, other)`	other - self	减法
`__rmul__(self, other)`	other * self	乘法
`__rtruediv__(self, other)`	other / self	除法
`__rfloordiv__(self, other)`	other // self	整数（地板除）
`__rmod__(self, other)`	other % self	求余(取模)
`__rpow__(self, other)`	other ** self	幂
`__rmatmul__(self, other)`	other @ self	矩阵乘法

二元反向算术运算符重载的方法格式:

```
def __rxxx__(self, other):
    ....
```

说明

1. 对于算术运算符， self 参数绑定运算符左侧的对象，other 绑定运算符右侧的对象。
2. 对于反向算术运算符， self 参数绑定运算符右侧的对象，other 绑定运算符左侧的对象。

示例

没有重载反向算数运算符的 Vector3D 类在运行 `v4 = 2 * v1` 时报错。如

```
# 反向算术运算符重载 示例

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __mul__(self, other):
        return Vector3D(self.x * other, self.y * other, self.z * other)

v1 = Vector3D(1, 2, 3)
v2 = Vector3D(4, 5, 6)
print("v1:", v1)
print("v2:", v2)
```

```
v4 = v1 * 2
print("v4:", v4)  # Vector3D(2, 4, 6)
v4 = 2 * v1  # 等同于 2.__mul__(v1),但 int 类型不支持此操作。报错
print("v4:", v4)  # Vector3D(2, 4, 6)
```

运行结果

```
v1: Vector3D(1,2,3)
v2: Vector3D(4,5,6)
v4: Vector3D(2,4,6)
Traceback (most recent call last):
  File "/Users/weimz/Desktop/
chapter_31/02_reverse_arithmetis_operator_overload.py", line 27, in <module>
    v4 = 2 * v1  # 等同于 2.__mul__(v1),但 int 类型不支持此操作。报错
         ~~^~~~
TypeError: unsupported operand type(s) for *: 'int' and 'Vector3D'
```

重载反向算数运算符的方法 `__rmul__` 后，运行结果正常。如:

```
# 反向算术运算符重载 示例

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __mul__(self, other):
        return Vector3D(self.x * other, self.y * other, self.z * other)

    def __rmul__(self, other):
        return self.__mul__(other)

v1 = Vector3D(1, 2, 3)
v2 = Vector3D(4, 5, 6)
print("v1:", v1)
print("v2:", v2)
v4 = v1 * 2
print("v4:", v4)  # Vector3D(2, 4, 6)
v4 = 2 * v1  # 调用 v1.__rmul__(2), 结果正确。
print("v4:", v4)  # Vector3D(2, 4, 6)
```

运行结果

```
v1: Vector3D(1,2,3)
v2: Vector3D(4,5,6)
```

```
v4: Vector3D(2,4,6)
v4: Vector3D(2,4,6)
```

3. 增强赋值算术运算符重载

增强赋值算术运算符重载的方法

方法名和参数	运算符和表达式	说明
`__iadd__(self, other)`	self += other	加法
`__isub__(self, other)`	self -= other	减法
`__imul__(self, other)`	self *= other	乘法
`__itruediv__(self, other)`	self /= other	除法
`__ifloordiv__(self, other)`	self //= other	整数（地板除）
`__imod__(self, other)`	self %= other	求余(取模)
`__ipow__(self, other)`	self **= other	幂
`__imatmul__(self, other)`	self @= other	矩阵乘法

示例

```
# 增强赋值算术运算符重载 示例
# http://weimingze.com

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __mul__(self, other):
        return Vector3D(
            self.x * other, self.y * other,
            self.z * other)

    def __imul__(self, other):
        self.x *= other
        self.y *= other
        self.z *= other
        return self
```

```
v1 = Vector3D(1, 2, 3)
print("v1:", v1)
print(id(v1))
v1 *= 3  # v1.__imul__(3)  # v1 = v1.__mul__(3)
print('v1:', v1)
print(id(v1))
```

运行结果

```
v1: Vector3D(1,2,3)
4358500240
v1: Vector3D(3,6,9)
4358500240
```

去掉 `__imul__` 方法，程序运行结果相同，只是ID会有变化。此时 v1 *= 3 后 v1绑定的不是原对象。

示例

```
# 增强赋值算术运算符重载 示例
# http://weimingze.com

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __mul__(self, other):
        return Vector3D(
            self.x * other, self.y * other,
            self.z * other)

v1 = Vector3D(1, 2, 3)
print("v1:", v1)
print(id(v1))
v1 *= 3  # v1.__imul__(3)  # v1 = v1.__mul__(3)
print('v1:', v1)
print(id(v1))
```

运行结果

```
v1: Vector3D(1,2,3)
4404751936
v1: Vector3D(3,6,9)
4406038032
```

增强赋值算术运算符的运算规则说明

以增强赋值算术运算符 `x += y` 为例，此运算符会优先调用 `x.__iadd__(y)` 方法，如果没有 `__iadd__()` 方法时会将增强赋值运算拆解为 `x = x + y`,然后调用 `x = x.__add__(y)` 方法；如果再不存在 `__add__()` 方法则会触发 TypeError 异常。

其它增强赋值算术运算符也具有相同的规则。

问题

以下两个程序打印结果是什么？为什么不一样？

f1 函数传入列表的程序运行：

```
# ---  以下用列表 ---
L = [1, 2, 3]
def f1(lst):
    lst += [4, 5, 6]
f1(L)
print(L)  # [1, 2, 3, 4, 5, 6]
```

f1 函数传入元组的程序运行：

```
# ---  以下用元组 ---
L = (1, 2, 3)
def f1(lst):
    lst += (4, 5, 6)
f1(L)
print(L)  # (1, 2, 3)
```

4. 比较运算符的重载

运算符和方法

方法名	运算符和表达式	说明
`__lt__(self, other)`	self < other	小于
`__le__(self, other)`	self <= other	小于等于
`__gt__(self, other)`	self > other	大于
`__ge__(self, other)`	self >= other	大于等于
`__eq__(self, other)`	self == other	等于
`__ne__(self, other)`	self != other	不等于

比较运算符通常返回布尔值 True 或 False。

示例

```
# 比较运算符重载 示例
# http://weimingze.com

class Vector1D:
    '''用于描述一维向量的类! '''
    def __init__(self, x):
        self.x = x

    def __repr__(self):
        return f'Vector1D({self.x})'

    def __lt__(self, other):  # lt : little than
        # print('__lt__: 小于号运算符重载')
        return self.x < other.x

    def __le__(self, other):  # le : little equal
        # print('__le__: 小于等于号运算符重载')
        return self.x <= other.x

    def __gt__(self, other):
        # print('__gt__: 大于号运算符重载')
        return self.x > other.x

    def __ge__(self, other):
        # print('__ge__: 大于等于号运算符重载')
        return self.x >= other.x

    def __eq__(self, other):
        # print('__eq__: 等于号运算符重载')
        return self.x == other.x

    def __ne__(self, other):
        # print('__ne__: 不等于号运算符重载')
        return self.x != other.x
```

```
v1 = Vector1D(100)
v2 = Vector1D(200)
print("v1:", v1)
print("v2:", v2)
print(v1 < v2)
print(v1 <= v2)
print(v1 > v2)
print(v1 >= v2)
print(v1 == v2)
print(v1 != v2)
```

运行结果

```
v1: Vector1D(100)
v2: Vector1D(200)
True
True
False
False
False
True
```

比较运算符重载说明

当没有与之相对的方法时，则采用对应的方法取值后再取非后返回。

方法名	运算符	对应方法名	对应运算符
`__lt__(self, other)`	<	`__gt__(self, other)`	>
`__le__(self, other)`	<=	`__ge__(self, other)`	>=
`__eq__(self, other)`	==	`__ne__(self, other)`	!=

> 当既没有 `__eq__(self, other)` 方法，也没有 `__ne__(self, other)` 方法时，将判断两个对象的ID是否相同，如果相同返回True,如果不同返回False。

详细说明

1. 大于运算符(>) 的重载方法是__gt__,当没有__gt__方法时，将调用__lt__方法获取值后取非(not),如果再没有__lt__方法会触发TypeError类型错误。

2. 小于运算符(<) 的重载方法是__lt__,当没有__lt__方法时，将调用__gt__方法获取值后取非(not),如果再没有__gt__方法会触发TypeError类型错误。
   - 小于运算符的重载与大于运算符的重载相反；

3. 大于等于运算符(>=) 的重载方法是__ge__,当没有__ge__方法时，将调用__le__方法获取值后取非(not),如果再没有__le__方法会触发TypeError类型错误。

4. 小于等于运算符(<=) 的重载方法是__le__,当没有__le__方法时，将调用__ge__方法获取值后取非(not),如果再没有__ge__方法会触发TypeError类型错误。
   - 小于等于运算符的重载与大于等于运算符的重载相反；

5. 等于运算符(==) 的重载方法是__eq__,当没有__eq__方法时，将判断两个运算符的ID是否相同，如果相同返回True,如果不同返回False。

6. 不等于运算符(!=) 的重载方法是__ne__,当没有__ne__方法时，将调用__eq__方法获取值后取非(not),如果再没有__eq__方法，则判断两个运算符的ID是否相同，如果不同返回True,如果相同返回False。

5. 位运算符重载

位运算符重载

运算符和方法

方法名	运算符和表达式	说明
`__invert__(self)`	~ self	取反(一元运算
`__and__(self, other)`	self & other	位与
`__or__(self, other)`	self | other	位或
`__xor__(self, other)`	self ^ other	位异或
`__lshift__(self, other)`	self << other	左移
`__rshift__(self, other)`	self >> other	右移

反向位运算符重载

运算符和方法

方法名	运算符和表达式	说明
`__rand__(self, other)`	other & self	位与
`__ror__(self, other)`	other | self	位或
`__rxor__(self, other)`	other ^ self	位异或
`__rlshift__(self, other)`	other << self	左移
`__rrshift__(self, other)`	other >> self	右移

增强赋值位运算符重载

运算符和方法

方法名	运算符和表达式	说明
`__iand__(self, other)`	self &= other	位与
`__ior__(self, other)`	self |= other	位或
`__ixor__(self, other)`	self ^= other	位异或
`__ilshift__(self, other)`	self <<= other	左移
`__irshift__(self, other)`	self >>= other	右移

示例

```
# 位运算符重载 示例
# http://weimingze.com

class Vector1D:
    '''用于描述一维向量的类! '''
    def __init__(self, x):
        self.x = x

    def __repr__(self):
        return f'Vector1D({self.x})'

    def __and__(self, other):  # &
        print('__and__: 位与运算符重载')

    def __or__(self, other):  # |
        print('__or__: 位或运算符重载')

    def __xor__(self, other):  # ^
        print('__xor__: 位异或运算符重载')
```

```
    def __lshift__(self, other):  # <<
        print('__lshift__: 左移运算符重载')

    def __rshift__(self, other):  # >>
        print('__rshift__: 右移运算符重载')

v1 = Vector1D(100)
v2 = Vector1D(200)
v1 & v2
v1 | v2
v1 ^ v2
v1 << v2
v1 >> v2
```

运行结果

```
__and__: 位与运算符重载
__or__: 位或运算符重载
__xor__: 位异或运算符重载
__lshift__: 左移运算符重载
__rshift__: 右移运算符重载
```

6. 一元运算符重载

一元运算符重载

运算符和方法

方法名	运算符和表达式	说明
`__neg__(self)`	-self	负号
`__pos__(self)`	+self	正号
`__invert__(self)`	~self	取反

一元运算符重载的方法的格式:

```
class 类名:
    def __xxx__(self):
        ...
```

> 只有一个参数绑定要运算的数据自身。

示例

```
# 一元运算符重载 示例

class Vector1D:
    '''用于描述一维向量的类! '''
    def __init__(self, x):
        self.x = x

    def __repr__(self):
        return f'Vector1D({self.x})'

    def __neg__(self):  # -
        print('__neg__: 负号运算符重载')
        return Vector1D(-self.x)

    def __pos__(self):  # +
        print('__pos__: 正号运算符重载')
        return Vector1D(self.x)

    def __invert__(self):  # ~
        print('__invert__: 取反运算符重载')
        return Vector1D(0)

v1 = Vector1D(100)
v2 = -v1
print('v2:', v2)
v2 = +v1
print('v2:', v2)
v2 = ~v1
print('v2:', v2)
```

7. 成员检测运算符重载

成员检测运算符有两个:

1. in运算符。
2. not in 运算符。

> 这两个运算符互为布尔非的关系。

成员检测运算符重载只有一个对应方法: `__contains__`。

成员检测运算符重载方法格式

```
class 类名:
    def __contains__(self, item):
        ...
```

运算符和方法

方法名	运算符和表达式	说明
`__contains__(self, item)`	item in self	成员检测运算

> `item not in self` 运算等同于 `not (item in self)` 运算。

示例

```
# 成员检测运算符重载 示例

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __contains__(self, item):
        ''' in 运算符的重载方法 '''
        print("item:", item)
        if item == self.x or item == self.y or item == self.z:
            return True
        return False

v1 = Vector3D(1, 2, 3)
print('v1:', v1)
print(3 in v1)
print(100 not in v1)
```

运行结果

```
v1: Vector3D(1,2,3)
item: 3
True
item: 100
True
```

成员检测运算符重载说明

- 如果类内有 `__contains__` 方法，返回 `__contains__` 方法的值。
- 如果类内没有 `__contains__` 方法，则用 `__iter__()` 方法取迭代器后遍历比较。
- 如果类内没有 `__iter__` 方法。则引发 TypeError 类型的错误。

8. 索引运算符重载

作用

让自定义类型的对象能够支持索引操作。

索引运算符重载的方法:

方法名	运算符和表达式	说明
`__getitem__`(self, item)	x = self[item]	索引取值
`__setitem__`(self, key, value)	self[key] = value	索引的赋值
`__delitem__`(self, key)	del self[key]	del语句删除索引

示例

```
# 索引运算符重载 示例
# http://weimingze.com

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __getitem__(self, item):
        print('索引取值, item:', item)

    def __setitem__(self, key, value):
        print(f'索引赋值, key: {key}, value: {value}')

    def __delitem__(self, key):
        print(f'删除索引, key: {key}')

v1 = Vector3D(1, 2, 3)
print('v1:', v1)
print('v1[0]:', v1[0])
```

```
v1[1] = 200
print('v1:', v1)
del v1[100]
```

运行结果

```
v1: Vector3D(1,2,3)
索引取值, item: 0
v1[0]: None
索引赋值, key: 1, value: 200
v1: Vector3D(1,2,3)
删除索引, key: 100
```

9. slice对象和切片重载

什么是切片对象

切片对象（slice）是一种用于从序列类型（如列表、字符串、元组等）中提取子序列的操作。它通过指定 start、stop 和 step 参数来截取序列的一部分，

切片语法为：

```
序列 [start:stop:step]
```

示例：

```
lst = [1, 2, 3, 4, 5, 6, 7]
print(lst[1:6:2]  # 其中 1:6:2 就是创建切片对象
```

切片对象的创建

slice函数

函数	说明
slice(start=None, stop=None, step=None)	创建一个slice切片对象。

slice 对象的属性

- start 切片的起始值，默认为None
- stop 切片的终止值，默认为None

- step 切片的步长，默认为None

切片运算符重载

作用

让自定义类型的对象能够支持切片操作。

索引运算符重载的方法:

方法名	运算符和表达式	说明
`__getitem__(self, item)`	x = self[item]	切片取值
`__setitem__(self, key, value)`	self[key] = value	切片的赋值
`__delitem__(self, key)`	del self[key]	del语句删除切片

> 索引和切片运算符重载使用相同的方法。

示例

```
# 切片运算符重载 示例

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __getitem__(self, item):
        print('__getitem__, item:', item)
        if isinstance(item, int):
            print('正在执行索引取值操作:')
        elif isinstance(item, slice):
            print('正在执行切片取值操作:')
            print('起始值，start:', item.start)
            print('终止值，stop:', item.stop)
            print('步长，step:', item.step)

    def __setitem__(self, key, value):
        print(f'__setitem__, key: {key}, value: {value}')

    def __delitem__(self, key):
        print(f'__delitem__, key: {key}')
```

```
v1 = Vector3D(1, 2, 3)
print('v1:', v1)
# value = v1[1:100:2]
value = v1[::2]
```

运行结果

```
v1: Vector3D(1,2,3)
__getitem__, item: slice(None, None, 2)
正在执行切片取值操作:
起始值，start: None
终止值，stop: None
步长，step: 2
```

10. 省略号常量

省略号 Ellipsis 是 Python中的一个常量。它通常用于扩展索引和切片操作的功能。

省略号的字面值是 Ellipsis。省略号字面值也可以写成三个点 ...。

作用：

用于自定义的容器类型扩展索引和切片的用法。如：numpy 和 pandas 模块。

省略号示例

numpy 是数据处理的一个包，他是第三方模块，需要安装。

numpy安装方法:

```
pip3 install numpy
```

示例：

```
import numpy

arr4d = numpy.ones(100).reshape(5, 2, 5, 2)
new_arr = arr4d[1:, ..., 1]
print(new_arr)
# 或
new_arr2 = arr4d[1:, Ellipsis, 1]
print(new_arr)
```

示例

在自定义的类中使用 省略号常量

```
# 省略号常量 示例
# http://weimingze.com

class Vector3D:
    '''用于描述三维向量的类！，x,y,z分别是向量的坐标位置！'''
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __repr__(self):
        return f'Vector3D({self.x},{self.y},{self.z})'

    def __getitem__(self, item):
        print('__getitem__, item:', item)

    def __setitem__(self, key, value):
        print(f'__setitem__, key: {key}, value: {value}')

    def __delitem__(self, key):
        print(f'__delitem__, key: {key}')

v1 = Vector3D(1, 2, 3)
print('v1:', v1)
# v1[...]
# v1[Ellipsis]
# value = v1[1:, ..., 100]
# value = v1[1:, Ellipsis, 100]
```

上述程序改为如下：

```
v1 = Vector3D(1, 2, 3)
print('v1:', v1)
v1[...]
# v1[Ellipsis]
# value = v1[1:, ..., 100]
# value = v1[1:, Ellipsis, 100]
```

运行结果如下

```
v1: Vector3D(1,2,3)
__getitem__, item: Ellipsis
```

上述程序改为如下：

```
v1 = Vector3D(1, 2, 3)
print('v1:', v1)
# v1[...]
```

```
v1[Ellipsis]
# value = v1[1:, ..., 100]
# value = v1[1:, Ellipsis, 100]
```

运行结果如下

```
v1: Vector3D(1,2,3)
__getitem__, item: Ellipsis
```

上述程序改为如下：

```
v1 = Vector3D(1, 2, 3)
print('v1:', v1)
# v1[...]
# v1[Ellipsis]
value = v1[1:, ..., 100]
# value = v1[1:, Ellipsis, 100]
```

运行结果如下

```
v1: Vector3D(1,2,3)
__getitem__, item: (slice(1, None, None), Ellipsis, 100)
```

上述程序改为如下：

```
v1 = Vector3D(1, 2, 3)
print('v1:', v1)
# v1[...]
# v1[Ellipsis]
# value = v1[1:, ..., 100]
value = v1[1:, Ellipsis, 100]
```

运行结果如下

```
v1: Vector3D(1,2,3)
__getitem__, item: (slice(1, None, None), Ellipsis, 100)
```

11. 属性访问与控制运算符的重载

作用

对 Python 中的对象的属性的取值、赋值、删除属性进行管理，能够动态管理对象属性的取值和赋值。

属性访问与控制运算符的重载的方法用四个，分别是：
`__getattribute__`,`__getattr__`,`__setattr__`,`__delattr__`。

属性访问与控制运算符的重载的方法:

方法名	运算符和表达式	说明
`__getattribute__(self, item)`	x = self.item	动态检查 item 属性，并返回 item 的值。
`__getattr__(self, item)`	x = self.item	访问不存在的属性时调用此方法。
`__setattr__(self, key, value)`	self.key = value	设置属性时调用（包括初始化时的赋值）。
`__delattr__(self, key)`	del self.key	用 del 语句删除属性时调用，如: `del obj.key`

> 索引和切片运算符重载使用相同的方法。

`__getattribute__(self, item)` 方法详解

作用:

在访问任何属性时被调用（无论访问属性还是调用方法都会调用，不管属性是否存在）。

触发条件

所有属性访问（包括 obj.x、obj.method()）。

用途:

访问任何属性时调用，拦截访问控制并处理。

注意事项

内部访问属性时需显式调用父类的 `__getattribute__` 方法，如 `super().__getattribute__()` 来避免递归。

`__getattr__(self, item)` 方法详解

作用

当访问对象的不存在的属性时被调用。

触发条件

仅当普通属性查找（如实例字典、类继承链）失败时或 `__getattribute__` 方法引发 AttributeError 调用。

用途

动态生成属性、实现懒加载或友好的错误提示。

`__setattr__(self, key, value)` 方法详解

作用

在设置任何属性（包括初始化时的属性）时被调用。

触发条件

对属性的赋值操作（如 obj.x = y）。

用途

拦截属性赋值，实现验证、日志记录或自动触发操作。

注意事项

内部赋值需通过 `self.__dict__` 或 `super()` 避免递归。

`__delattr__(self, key)` 方法详解

作用

在删除属性时被调用（如 del obj.x）。

触发条件

执行 del 操作时。

用途

阻止删除关键属性或执行清理操作。

12. 特殊方法总结

运算符重载 回顾

运算符重载是指通过自定义类中的特殊方法来让自定义的类创建的对象可以使用运算符进行操作。

类相关的特殊方法

类别	方法名
对象创建、初始化和销毁	`__new__`, `__init__`, `__del__`
字符串表示形式	`__repr__`, `__str__`
数值转换	`__int__`, `__float__`, `__complex__`, `__bool__`
数值运算	`__abs__`, `__round__`
集合模拟	`__len__`, `__reversed__`
可迭代对象和迭代器	`__iter__`, `__next__`
函数调用	`__call__`
上下文管理	`__enter__`, `__exit__`
描述符协议	`__get__`, `__set__`, `__delete__`

运算符相关的特殊方法

类别	方法名
算术运算符	`__add__`, `__sub__`, `__mul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__pow__`, `__matmul__`
反向算术运算符	`__radd__`, `__rsub__`, `__rmul__`, `__rtruediv__`, `__rfloordiv__`, `__rmod__`, `__rpow__`, `__rmatmul__`
增强赋值算术运算符	`__iadd__`, `__isub__`, `__imul__`, `__itruediv__`, `__ifloordiv__`, `__imod__`, `__ipow__`, `__imatmul__`
比较运算符	`__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`
位运算符	`__and__`, `__or__`, `__xor__`, `__lshift__`, `__rshift__`

反向位运算符	`__rand__`, `__ror__`, `__rxor__`, `__rlshift__`, `__rrshift__`
增强赋值位运算符	`__iand__`, `__ior__`, `__ixor__`, `__ilshift__`, `__irshift__`
一元运算符	`__neg__`, `__pos__`, `__invert__`
索引和切片	`__getitem__`, `__setitem__`, `__delitem__`
属性访问与控制	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`

成员检测运算符	`__contains__`	

不能被重载的运算符

运算符	说明
and、or、not	布尔运算
is、is not	对象ID比较
=	赋值语句分隔符

查看列表 list 类的特殊方法

当我们拿到一个类，不知道这个类如何来用的时候。我们可以用它来创建一个对象，然后用交互模式的help()函数，就可以查出这个类有哪些函数或者是方法可以用了。

示例如下：

```
>>> lst = [1, 2, 3, 4]
>>> help(lst)
```

第三十二章、元类

1. 元类

什么是元类

元类是创建类的类。在 Python 中一切皆是对象，类也是对象，而创建类的类就是元类。

任何对象的 `__class__` 特殊属性都绑定创建此对象的类。类也是如此。

示例:

```
# 元类 示例

class Dog:
    pass

dog1 = Dog()

print(dog1)
print(dog1.__class__)  # dog1绑定的对象的类是 Dog
print(Dog.__class__)  # Dog类的类是 type
print(type.__class__)  # type 类的类还是 type
```

运行结果

```
<__main__.Dog object at 0x106b02e40>
<class '__main__.Dog'>
<class 'type'>
<class 'type'>
```

class 语句创建类

特点:

1. 适用于大多数场景。特点是常用且直观。
2. 默认使用元类 type 创建类。
3. 支持继承、方法定义、属性初始化等标准特性。

创建类的三种方式:

1. 用 class 语句来创建新的类。最常用的创建方式。

2. 用 type 函数来直接创建新的类。用于需要动态生成类的场景。
3. 用 继承 type 类来创建自定义元类，再由此元类创建类。用于框架定制等功能。

2. type 类

type 类是一切类的基础元类，所有的类都直接或间接的由 type 类创建。

我们可以用 type 类的构造函数直接创建类。

作用

type函数创建类主要用于动态的来生成类。

type函数

函数	说明
type(obj)	返回创建 obj 的类。
type(name, bases, namespace)	用所给参数动态创建一个新的类。返回新的类。

参数说明：

- name：类名字符串。
- bases：基类的元组
- namespace：成员的字典

示例

```
# type 函数创建类 示例

class Dog:
    home = '地球'
    def speak(self):
        return print('旺! ')

# 创建一个同 `class Dog` 同样的类

MyDog = type('MyDog', (object,), {'home': '地球', 'speak': lambda self: print('旺旺! ')})
dog1 = MyDog()
print(dog1.__class__)
```

```
dog1.speak()
print(Dog.home)
```

type函数创建类的示例1：

```
class A:
    a = 100
# 等同于
A = type('A', (object,), dict(a=100))
```

type函数创建类的示例2：

```
class B():
    b = 100
# 等同于
B = type('B', (), dict(b=100))
```

type函数创建类的示例3：

```
class C:
    c = 100
    def hello(self):
        return print("hello")
# 等同于
C = type('C', (object,), dict(hello=lambda self: print("hello"), c=100))
```

type函数创建类的示例4：

```
class D:
    def fa(self):
        print("hello:", id(self))

# 等同于
def say_hello(self):
    print("hello:", id(self))

D = type('D', (), {"fa": say_hello})
```

type函数创建类的示例5：

```
class E:
    @classmethod
    def hello(cls):
        print("hello: ", type(cls))
```

```
# 等同于
@classmethod
def say_hello(cls):
    print("hello: ", type(cls))

E = type('E', (), {"hello": say_hello})
```

type函数创建类的示例6：

```
class F(list):
    pass
# 等同于
F = type('F', (list,), {})
```

3. 自定义元类

在 Python 语言中，用户可以自己定义元类，用户自定义的元类必须直接或间接的继承自 type 类。

作用

自定义元类通常用于控制此元类创建类的行为。实现自定义的功能。

语法

```
class 新元类(基础元类):
    ...
```

如:

```
class MyMetaClass(type):
    pass
```

说明

自定义元类的 可以通过覆盖 type 类的 `__new__` 和 `__init__` 方法来控制类在创建过程中的行为。

使用元类来创建新类

语法

```
class 新类(基类1, 基类2, ...,  metaclass=元类):
    ...
```

如:

```
class Animal(object, metaclass=MyMetaClass):
    home = '地球'
    def print_home(self):
        print(self.__class__.home)
```

说明

- 如果一类个默认没有指定 metaclass，则这个类直接由 type 类来创建。
- 如果一类指定了 metaclass，则这个类直接由 metaclass 指定的类来创建。
- 因为元类都直接或间接的继承自 type 类，因此所有用自定义元类创建的新类都要调用 `type.__new__(cls, name, bases, namespace)` 来创建新类，即使用type函数来创建类。

元类的 `__new__` 和 `__init__` 方法

1. 元类的 `__new__` 方法在类定义时执行，用于创建类并指定类属性和方法等。
2. 元类的 `__init__` 方法在新类创建后执行，用来初始化类。但魏老师没有用到过这个方法，不常用。

示例

自定义元类 MyMetaClass，此类继承自 type。

```
class MyMetaClass(type):
    pass
```

用 MyMetaClass 创建一个新类 Animal，此类继承自object，用元类 MyMetaClass 创建。

```
class Animal(object, metaclass=MyMetaClass):
    pass
```

测试打印 创建 Animal 的类和 MyMetaClass 的类都是什么?

```
print('创建Animal类的类是:', Animal.__class__)
print('创建MyMetaClass类的类是:', MyMetaClass.__class__)
```

运行结果

```
创建Animal类的类是: <class '__main__.MyMetaClass'>
创建MyMetaClass类的类是: <class 'type'>
```

改写MyMetaClass 添加 `__new__` 和 `__init__` 方法。并打印传入的参数。如下：

```
class MyMetaClass(type):
    def __new__(cls, name, bases, namespace):
        print('MyMetaClass.__new__: name:', name, 'bases:', bases, 'namespace',
namespace)
        return super().__new__(cls, name, bases, namespace)

    def __init__(self, name, bases, namespace):
        print('MyMetaClass.__init__: name:', name, 'bases:', bases, 'namespace',
namespace)
        super().__init__(name, bases, namespace)


class Animal(object, metaclass=MyMetaClass):
    pass
```

运行结果如下：

```
MyMetaClass.__new__: name: Animal bases: (<class 'object'>,) namespace
{'__module__': '__main__', '__qualname__': 'Animal', '__firstlineno__': 15,
'__static_attributes__': ()}
MyMetaClass.__init__: name: Animal bases: (<class 'object'>,) namespace
{'__module__': '__main__', '__qualname__': 'Animal', '__firstlineno__': 15,
'__static_attributes__': ()}
```

可见，在创建类 Animal 时 调用了 `__new__` 和 `__init__` 方法。并传入了类 Animal的名称、继承元组和属性信息。

修改 Animal 类如下，添加了两个类属性。继续运行程序

```
class Animal(object, metaclass=MyMetaClass):
    home = '地球'
    def print_home(self):
        print(self.__class__.home)
```

运行结果如下：

```
MyMetaClass.__new__: name: Animal bases: (<class 'object'>,) namespace
{'__module__': '__main__', '__qualname__': 'Animal', '__firstlineno__': 15,
'home': '地球', 'print_home': <function Animal.print_home at 0x1091c45e0>,
'__static_attributes__': ()}
MyMetaClass.__init__: name: Animal bases: (<class 'object'>,) namespace
{'__module__': '__main__', '__qualname__': 'Animal', '__firstlineno__': 15,
```

```
'home': '地球', 'print_home': <function Animal.print_home at 0x1091c45e0>,
'__static_attributes__': ()}
```

可见 Animal 类中的两个类属性也会以键值对的形式传递给元类 的 `__new__` 和 `__init__` 的 第四个参数 namespace。

最后再创建一个Animal类的子类，在创建此类时依旧会调用元类 MyMetaClass 中的 `__new__` 和 `__init__` 方法。代码如下

```
class MyMetaClass(type):
    def __new__(cls, name, bases, namespace):
        print('MyMetaClass.__new__: name:', name, 'bases:', bases, 'namespace',
namespace)
        return super().__new__(cls, name, bases, namespace)

    def __init__(self, name, bases, namespace):
        print('MyMetaClass.__init__: name:', name, 'bases:', bases, 'namespace',
namespace)
        super().__init__(name, bases, namespace)


class Animal(object, metaclass=MyMetaClass):
    home = '地球'
    def print_home(self):
        print(self.__class__.home)

class Dog(Animal):
    pass

print('创建Animal类的类是:', Animal.__class__)
print('创建MyMetaClass类的类是:', MyMetaClass.__class__)
```

运行结果

```
MyMetaClass.__new__: name: Animal bases: (<class 'object'>,) namespace
{'__module__': '__main__', '__qualname__': 'Animal', '__firstlineno__': 15, 'hom
e': '地球', 'print_home': <function Animal.print_home at 0x1091c45e0>, '__static_
attributes__': ()}
MyMetaClass.__init__: name: Animal bases: (<class 'object'>,) namespace {'__modu
le__': '__main__', '__qualname__': 'Animal', '__firstlineno__': 15, 'home': '地
球', 'print_home': <function Animal.print_home at 0x1091c45e0>, '__static_attrib
utes__': ()}
MyMetaClass.__new__: name: Dog bases: (<class '__main__.Animal'>,) namespace {'_
_module__': '__main__', '__qualname__': 'Dog', '__firstlineno__': 21, '__static_
attributes__': ()}
MyMetaClass.__init__: name: Dog bases: (<class '__main__.Animal'>,) namespace
{'__module__': '__main__', '__qualname__': 'Dog', '__firstlineno__': 21, '__stat
ic_attributes__': ()}
创建Animal类的类是: <class '__main__.MyMetaClass'>
创建MyMetaClass类的类是: <class 'type'>
```